National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES
THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

NL-339 (r.86/06)

THE UNIVERSITY OF ALBERTA

Memory-Segmented Programmable Sequential Controllers

by

Werner Eusebio Fraga

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Electrical Engineering

EDMONTON, ALBERTA

Spring 1987

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-37745-3

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Werner Eusebio Fraga

TITLE OF THESIS: Memory-Segmented Programmable Sequential Controllers

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: Spring 1987

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) Werner Fraga

PERMANENT ADDRESS:

7107 SASKATCHEWAN DRIVE

EDMONTON, ALBERTA

CANADA

DATED 12 AUG 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled Memory-Segmented Programmable Sequential Controllers submitted by Werner Eusebio Fraga in partial fulfilment of the requirements for the degree of Master of Science.

.....Keith A. Sharman.....

Supervisor

.....J. Khoo.....

.....Ernest Oz.....

Date.....12 Aug 1986.....

To Cindy

ABSTRACT

A programmable sequential controller may be used to implement industrial control processes which are described by finite state machines. Different methods of design of such a controller are examined and compared. The memory segmentation technique, which involves the division of the finite state program memory into four distinct data groups, was found to have several advantages over other designs: efficient use of program memory; high speed of operation; and simple programming language. The technique was implemented using a semiparallel design which processes the logic-related program data on the first clock cycle and the output-related data on the second clock cycle.

The controller circuit was designed as a four-bit slice cascadable integrated circuit. The design was layed out with the intent of achieving a square, compact circuit with the minimum amount of signal routing. Each section of the circuit was designed from the functional level to the gate level to the transistor level. The circuit was simulated at the transistor level to verify its operation and to determine propagation delays.

After fabrication, the circuit was tested for design and fabrication faults using microprocessor-based algorithms and both implicit and explicit testing techniques. Problems were detected in the output pads, register latches, and propagation delays, but with temporary corrections, the circuit operated satisfactorily. Design modifications were proposed to improve the circuit operation.

A low-level language was designed to further improve the programmability of

the controller. An assembler was written and tested to translate the low-level finite state machine programs into machine language programs to be run by the controller.

The author wishes to acknowledge the encouragement and assistance of his supervisor, Dr. Keith A. Stromsmoe. A research scholarship from the Faculty of Graduate Studies and Research, University of Alberta, and the technical support of the Canadian Microelectronics Corporation, are also recognised.

TABLE OF CONTENTS

Chapter		Page
I	PROGRAMMABLE SEQUENTIAL CONTROLLERS	1
	1.0 Introduction	1
	1.1 Controller design methods	5
	1.2 Memory segmentation techniques	11
II	INTEGRATED CIRCUIT DESIGN	22
	2.0 Introduction	22
	2.1 Design specifications	23
	2.2 System architecture	24
	2.3 Floor plan	30
	2.4 Logic design	39
	2.5 Gate design	59
	2.6 Circuit simulations	63
	2.7 Circuit layout	71
III	A LOW-LEVEL-LANGUAGE ASSEMBLER	78
	3.0 Introduction	78
	3.1 Assembler specifications	80
	3.2 Assembler design	83
IV	INTEGRATED CIRCUIT TEST	88
	4.0 Introduction	88
	4.1 Explicit testing	90
	4.2 Implicit testing	94

Chapter Page

4.3 Timing considerations 102

4.4 Design modifications 105

V CONCLUSION 109

REFERENCES 111

APPENDIX 1. SPICE circuit simulations 115

APPENDIX 2. PSC low-level language assembler program 122

APPENDIX 3. Assembler I/O 133

APPENDIX 4. PSC circuit test programs 137

LIST OF TABLES

Table		Page
1	Microprogramming required to implement the FSM of Figure 1 using PLA and PROM circuits	7
2	Memory segmentation microprogram for the FSM of Figure 1	13
3	Transition tables for lowest address bits for serial and and semiparallel segmentation designs	16
4	Number of pins for various IC configurations	25
5	Comparison of different IC configuration characteristics	28
6	Truth table and Karnaugh map for five-input multiplexer	45
7	Truth table and Karnaugh map for low address bit C1	49
8	Truth table and Karnaugh map for high address bit SB	51
9	Truth table and Karnaugh map for initial carry-in Co	54
10	Truth table and Karnaugh map for return address latch clock	56
11	Truth table and Karnaugh map for present address latch clock	57
12	Worst case rise and fall times and propagation delays for various simulated circuits	65
13	Effect of PSC language commands on three external PROMs	82
14	Comparison of measured and predicted propagation delays within the controller IC	103

LIST OF FIGURES

Figure		Page
1	Three-state Mealy machine	2
2	FSM for traffic control system	3
3	Programmable logic array (PLA)	6
4	PLA implementation of FSM of Figure 1	8
5	PROM implementation of FSM of Figure 1	10
6	Memory segmentation flow chart	12
7	Controller design using serial memory segmentation	15
8	Controller design using parallel memory segmentation	17
9	Controller design using semiparallel memory segmentation	19
10	Block diagram of cascaded PSC chips	27
11	Initial floor plan	31
12	Logic block floor plan	32
13	Address block floor plan	34
14	Control block floor plan	36
15	Final floor plan	37
16	Gate diagrams of data registers using D flip-flop and transmission gates	40
17	Edge-triggered register circuits	42
18	Gate diagrams of two-input and five-input multiplexers	44
19	Gate diagram of one-bit half adder	47
20	Gate diagrams of low address bit C1 and its inverse, $\overline{C1}$	50

Figure Page

21 Gate diagrams of high address bit SB and its inverse, $\overline{SB}$ 53

22 Transistor diagrams of inverter, NAND, and NOR gates 60

23 Transistor diagrams of EXOR gate and data register 61

24 Stick diagrams of inverter 72

25 GETADDRESS branch of assembler program structure tree 84

26 TRANSLATE branch of assembler program structure tree 85

27 Pin-out diagram of PSC IC 89

28 Modified clock circuit to slow rise and fall times 93

29 FSMs for I/O latching tests 96

30 FSM for subroutine/interrupt test 97

31 BDM for greatest-common-divisor circuit test 99

32 Block diagram of external circuitry for GCD test 101

33 Carry look-ahead generation for four-bit half-adder 108

LIST OF PHOTOGRAPHIC PLATES

Plate		Page
1.	Integrated circuit layout	76

LIST OF ABBREVIATIONS

BDM: binary decision machine.

CK: the clock signal.

CI: the chip carry-in signal.

Co: the universal initial carry-in signal.

CO: the chip carry-out signal.

CS: the subroutine call signal (when SB is low) and the return-from-subroutine signal (when SB is high).

C1: the low address bit which is used to address either the input mask and input condition words (C1 low), or the new address and output words (C1 high).

FSM: finite state machine.

IACK: the interrupt acknowledge signal (active low).

IC: integrated circuit.

IM: the interrupt mask (active low).

EINT: the external interrupt signal (active high).

$\overline{\text{INT}}$: the internal interrupt signal (active low).

I/O: input/output.

$\overline{\text{L}}$: the universal load signal (active low).

MHz: megaHertz (millions of cycles per second).

MUX2: the two-to-one multiplexer which permits either the present address or the return address to be stored in the return address register.

MUX5: the five-to-one multiplexer which permits one of reset, interrupt, present address, new address, or return address to be stored in the present address register.

NI: the load signal produced by the previous chip.

NO: the load signal produced by the chip.

ns: nanosecond

PLA: programmable logic array.

PROM: programmable read-only memory.

PSC: programmable sequential controller.

$\overline{\text{RES}}$: the reset signal (active low).

SB: the high address signal which is used to address either the main program (SB low) or subroutines (SB high).

I PROGRAMMABLE SEQUENTIAL CONTROLLERS

1.0 Introduction

Sequentially-controlled processes, from vending machines to cement mixers, can often be described by finite state machines. A finite state machine (FSM) is a system with a finite number of states or nodes, and a finite number of input-dependent paths from one node to another.

Figure 1 shows a simple three-state machine. Control will change from state A to state B when the input I_1 is high; if I_1 is low then control remains in state A.

The state of an FSM can be determined by the outputs produced by the FSM. In the FSM shown in Figure 1, the outputs are a function of the transition from one state to another. For example, the transition from A to B with I_1 high results in the output O_1 being set high. This type of FSM is called a Mealy machine. If the outputs are a function of the present state instead of the path between states, the system is called a Moore machine.

An example of a controlled process is a traffic control system consisting of two sets of traffic lights. The simplest version of this system is shown in Figure 2. The only input is a signal from a timer which causes a change of lights. There are ten outputs: signals to two sets of three lights, and four commands to load the programmable timer.

There is no problem in designing a digital controller to implement such a fixed process. A state machine having any number of states from $2^{n-1}+1$ to 2^n would require n registers to store the present state, and a set of logic circuits to control changes of state and produce the desired outputs. The principal

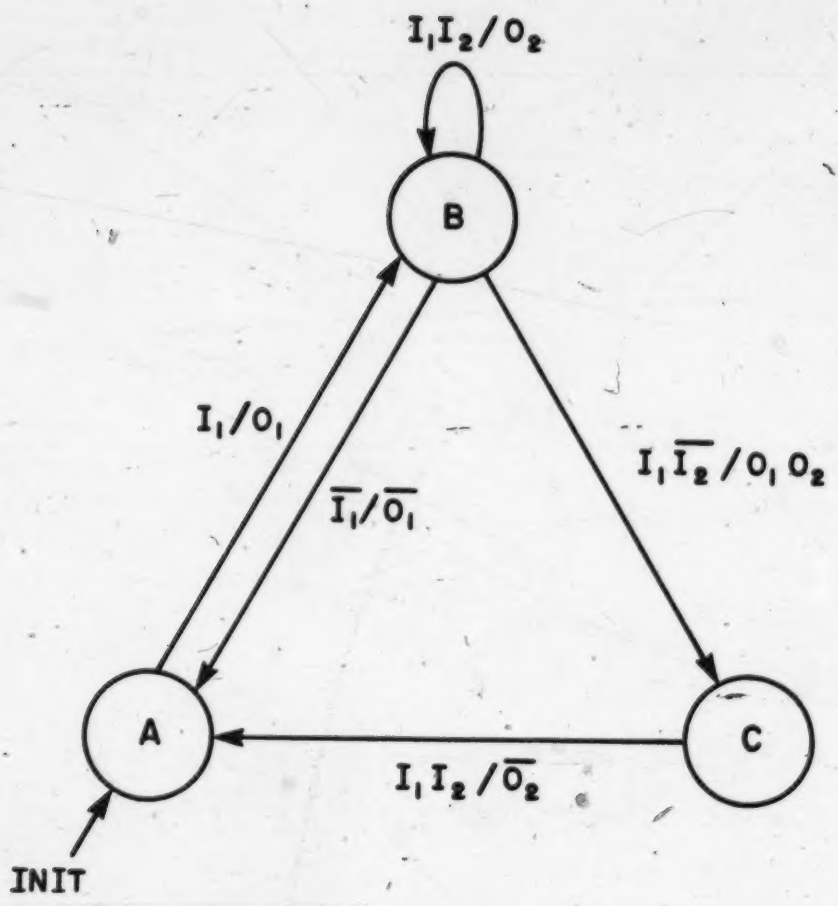


Figure 1. Three-state Mealy machine.

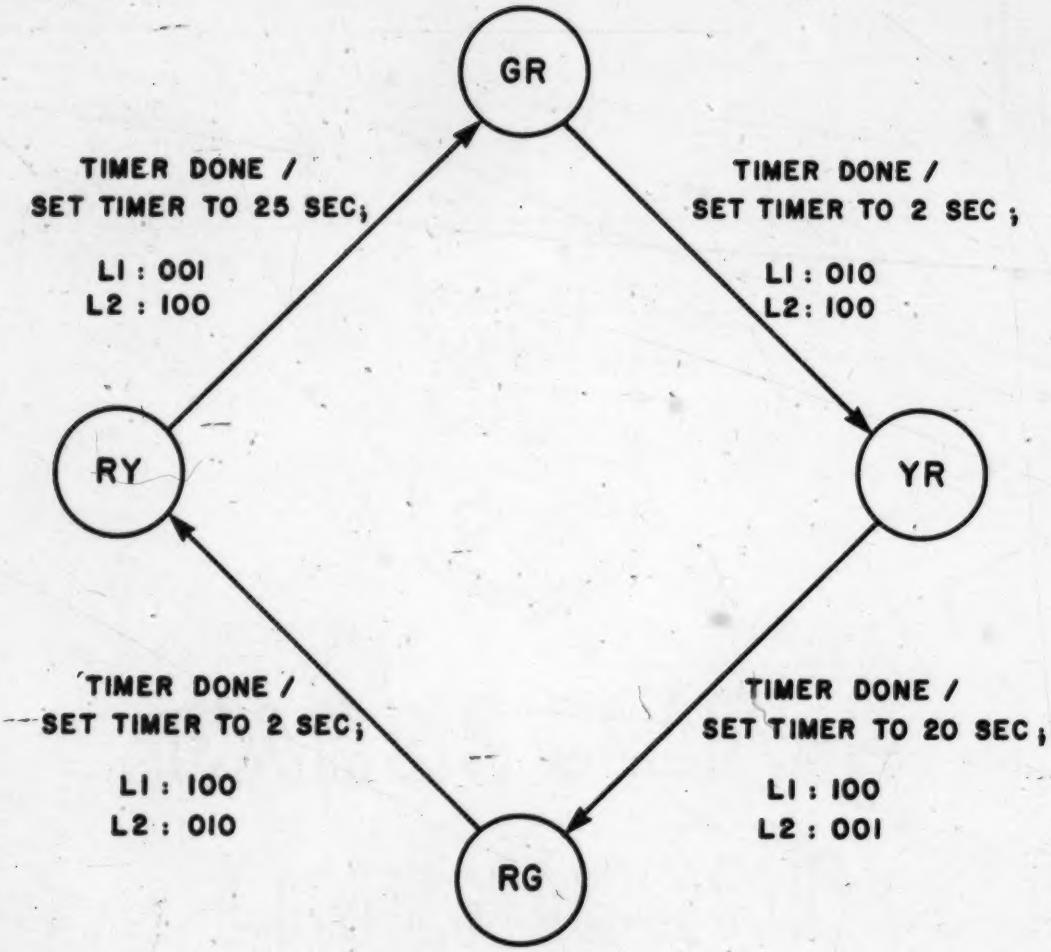


Figure 2. FSM for traffic control system.

objectives of the design are to minimise the size of the design and improve the speed of the circuit.

In many circumstances, however, the process will not be fixed. The user may want to change certain parameters in the process, add new states, or change the outputs from time to time. For example, the traffic control system may require different timer settings due to changes in traffic patterns, or in order to prolong the yellow (clear intersection) state.

In such cases, a programmable sequential controller (PSC) which can be programmed to implement any finite state machine (within some size restriction) is needed. The PSC could be programmed directly from a state graph such as the one shown in Figure 1. There are several methods of designing a PSC; the ideal PSC will have a minimal design, be easy to program and install, use as little program memory as possible, and operate at high speeds.

1.1 Controller design methods

A programmable logic array (PLA) [32,38] may be used to implement a programmable sequential controller. A PLA is simply a two-level logic network (Figure 3): AND gates on the first level and OR gates on the second level. Any desired function is written in its "sum of products" form and the necessary connections on the PLA are made. This makes the design fairly simple (although slow) to derive and very easy to implement. Field programmable logic arrays (FPLA), which can be programmed after manufacture, make the controller reprogrammable.

The PSC is designed by determining the next-state as a function of the present state and inputs. A register is used to synchronously latch the next state. This design has two advantages: it will require only one clock cycle to change states and it will have a minimisable circuit because the "don't care" terms (such as inputs that do not cause a branch from some state) do not affect the design. However, there are several disadvantages to the FPLA-based design. As the number of states and transitions increases, the design step required to program the PSC becomes more complex and the logic array becomes too large to be practical.

Table 1.a shows the design step required to program the FSM of Figure 1. The PLA implementation is shown in Figure 4. For a small FSM of only three states, two inputs, and two outputs, the PLA requires 22 connections to be made on a 7 by 12 array. For a much larger state machine, the number of connections and size of the array needed to store the FSM program would be detrimental.

Another method of design uses a programmable read-only memory (PROM) as the program-storage element of the circuit [13,14]. This eliminates the logic section of the PLA circuit. Each combination of a possible input word and

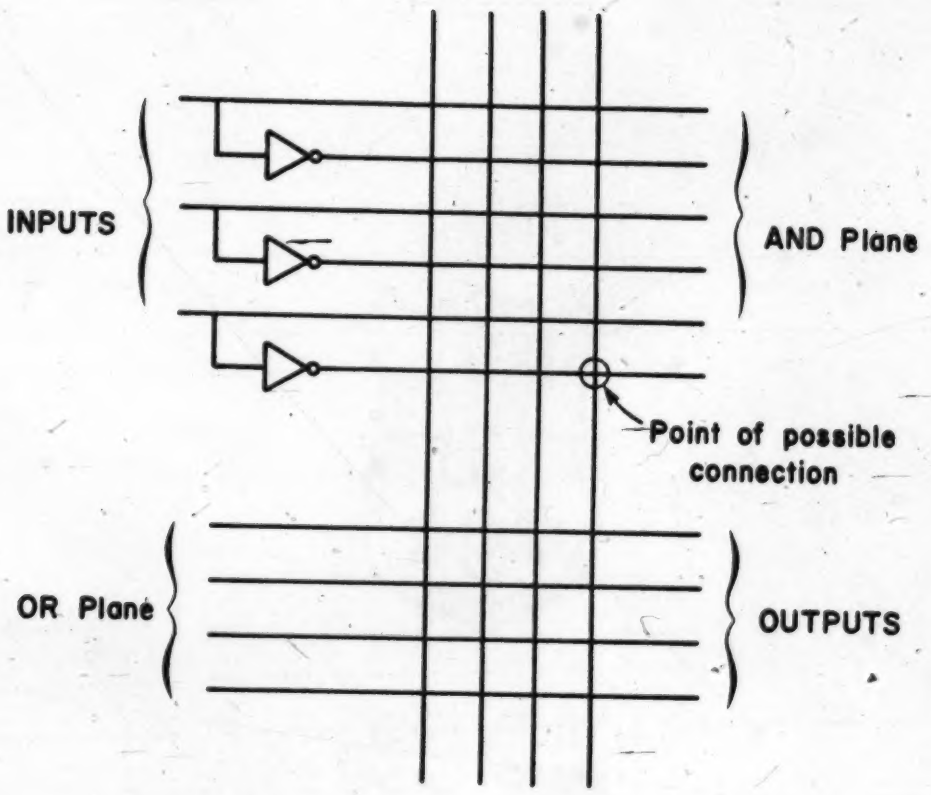


Figure 3. Programmable logic array (PLA).

PRESENT STATE X Y	NEXT STATE $X^+ Y^+$ DUE TO INPUT I1 I2				OUTPUT O1 O2 DUE TO INPUT I1 I2			
	00	01	10	11	00	01	10	11
A: 00	00	00	01	01	??	??	1?	1?
B: 01	00	00	10	01	0?	0?	11	?1
C: 10	10	10	10	00	??	??	??	?0

$$X^+ = (X)(\overline{I1}) + (X)(\overline{I2}) + (Y)(I1)(\overline{I2})$$

$$Y^+ = (\overline{X})(\overline{Y})(I1) + (\overline{X})(I1)(I2)$$

$$O1 = I1 \qquad O2 = \overline{X}$$

a) PLA design step, with ? as don't care

PRESENT STATE	PROM ADDRESS BINARY	HEX	PROM CONTENTS BINARY	HEX
A	0000	0	0000	0
	0001	1	0000	0
	0010	2	0110	6
	0011	3	0110	6
B	0100	4	0000	0
	0101	5	0000	0
	0110	6	1011	B
	0111	7	0101	5
C	1000	8	1000	8
	1001	9	1000	8
	1010	A	1000	8
	1011	B	0000	0

b) PROM program, using first two bits for present/next state and last two bits for input/output.

Table 1. Microprogramming required to implement the FSM of Figure 1 using a) PLA and b) PROM circuit

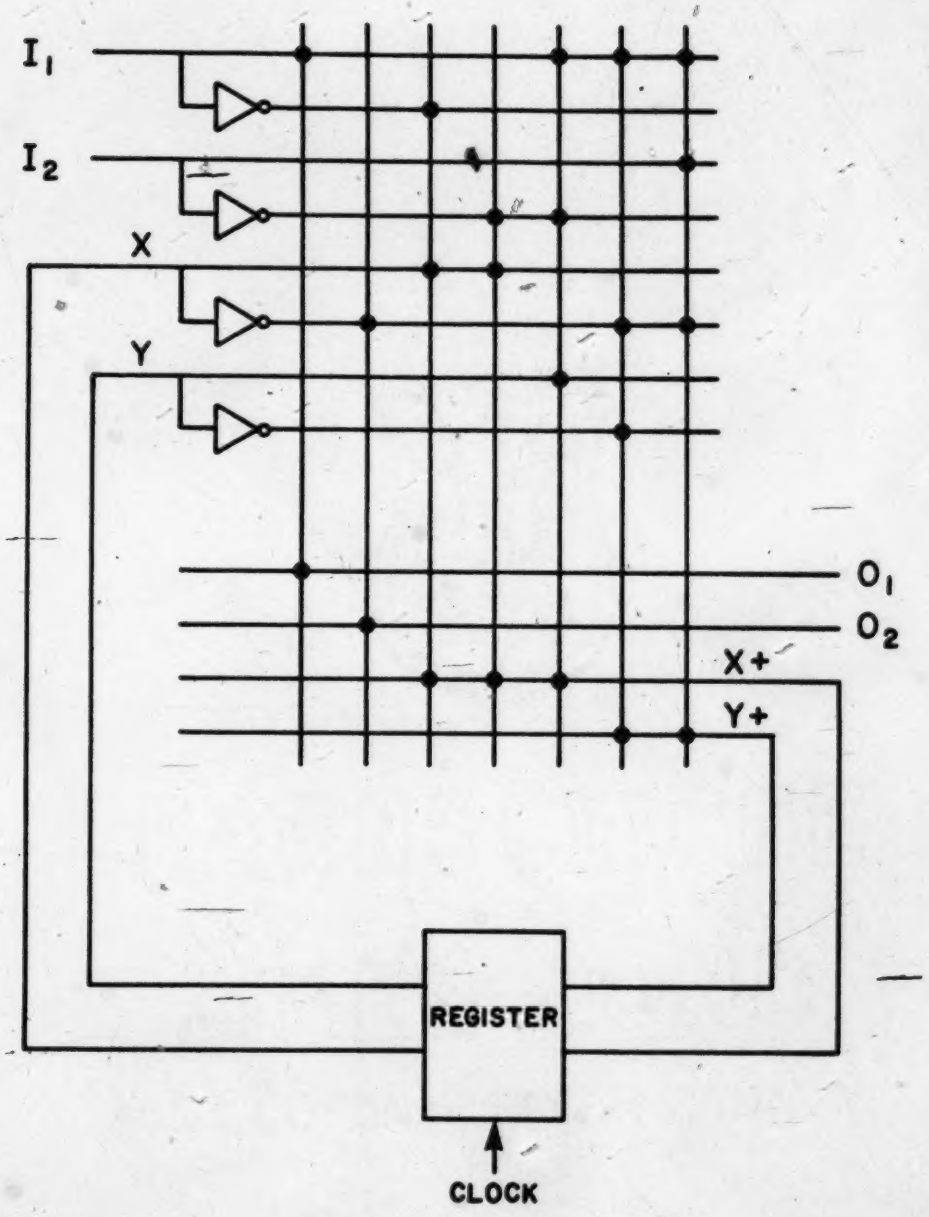


Figure 4. PLA implementation of FSM of Figure 1.

present state represents an address on the PROM. At this address are stored the next state in the state graph and the state/transition outputs. The next state information is fed back to the PROM address bits through a synchronous (clocked) register.

As in the PLA design, this circuit produces a change of state in only one clock cycle. The maximum clock frequency will depend on the propagation delay of the register and the minimum access time of the PROM.

This design simplifies the microprogramming tremendously. The states and transition inputs and outputs are converted directly to binary words and the results are written on the PROM. Unlike the PLA design, there is no design step necessary to program the PSC.

However, the amount of program storage space necessary may be excessive. Every possible input, including those which do not affect the present state, must be represented in the PROM. If the state machine being implemented has N states and an input word of n bits, then N times 2^n words of memory are needed. Clearly, this will severely limit the maximum size and number of transitions of the FSM.

Table 1.b shows the PROM microprogram that will implement the FSM of Figure 1. The PROM-based circuit, as shown in Figure 5, requires only 16 nybbles of memory. However, an FSM with just ten states and byte-long inputs and outputs would use 2560 words of memory, with a word length of twelve bits.

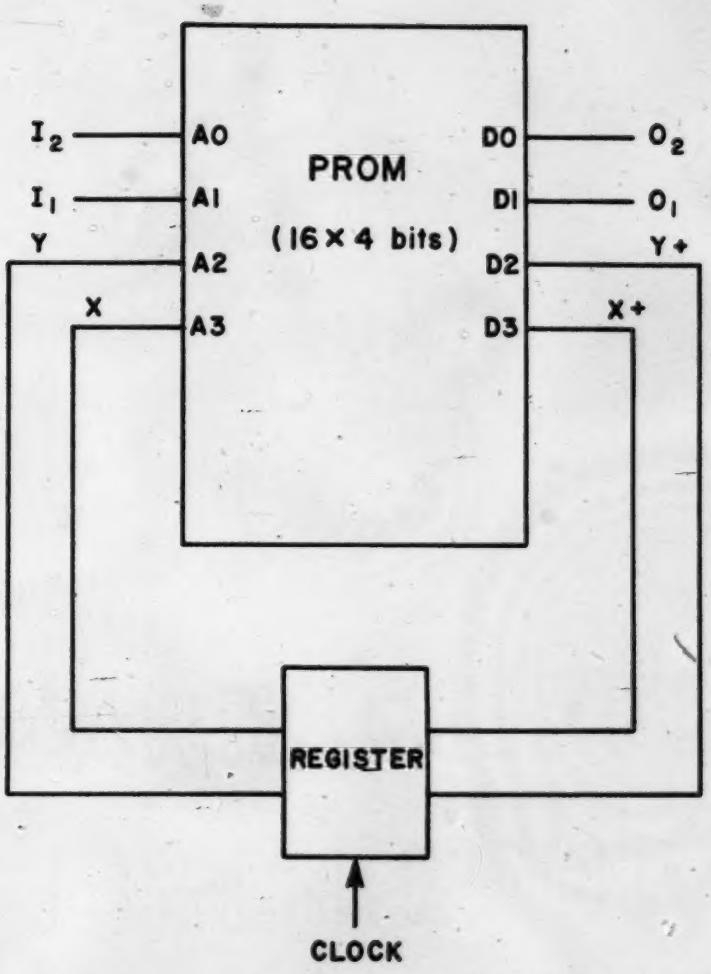


Figure 5. PROM implementation of FSM of Figure 1.

1.2 Memory segmentation techniques

What is needed is a PSC design which will have the speed of the PLA and PROM designs, the minimum-size design of the PLA circuit, and the micro-programmability of the PROM circuit, without using large amounts of program memory.

Memory segmentation [8] is a technique which involves the structuring of data into separate microinstruction fields. Each path or branch in an FSM may be described by five data fields: the present state or address, the next state or address, the input mask which selects the inputs relevant to the branch, the input condition upon which the branch should occur, and the outputs. A PSC may be designed which uses these data fields as a microprogram stored in PROM.

The algorithm for the memory segmentation design is quite simple (Figure 6). In any state (as specified by its address), the highest priority branch address, input mask, input condition, and outputs are loaded from PROM. The inputs are then masked (NORed) with the input mask and the result is compared (EXOR) to the input condition. If all bits agree (NAND all bits together and check for a zero), the branch takes place: the next state's address becomes the present state address. Otherwise the present address is incremented to access the data fields dealing with the next highest priority branch. The case in which control should remain in the same state because no successful branch has occurred is handled last: all inputs are masked to guarantee an unconditional branch back to the original address of the present state. This means that the data field for the present state is not needed, since it can be stored in the micro-program as the "next state" of lowest priority.

This algorithm requires four words of information to be stored in PROM for each branch in the FSM. Table 2 shows the microprogram implementing the state

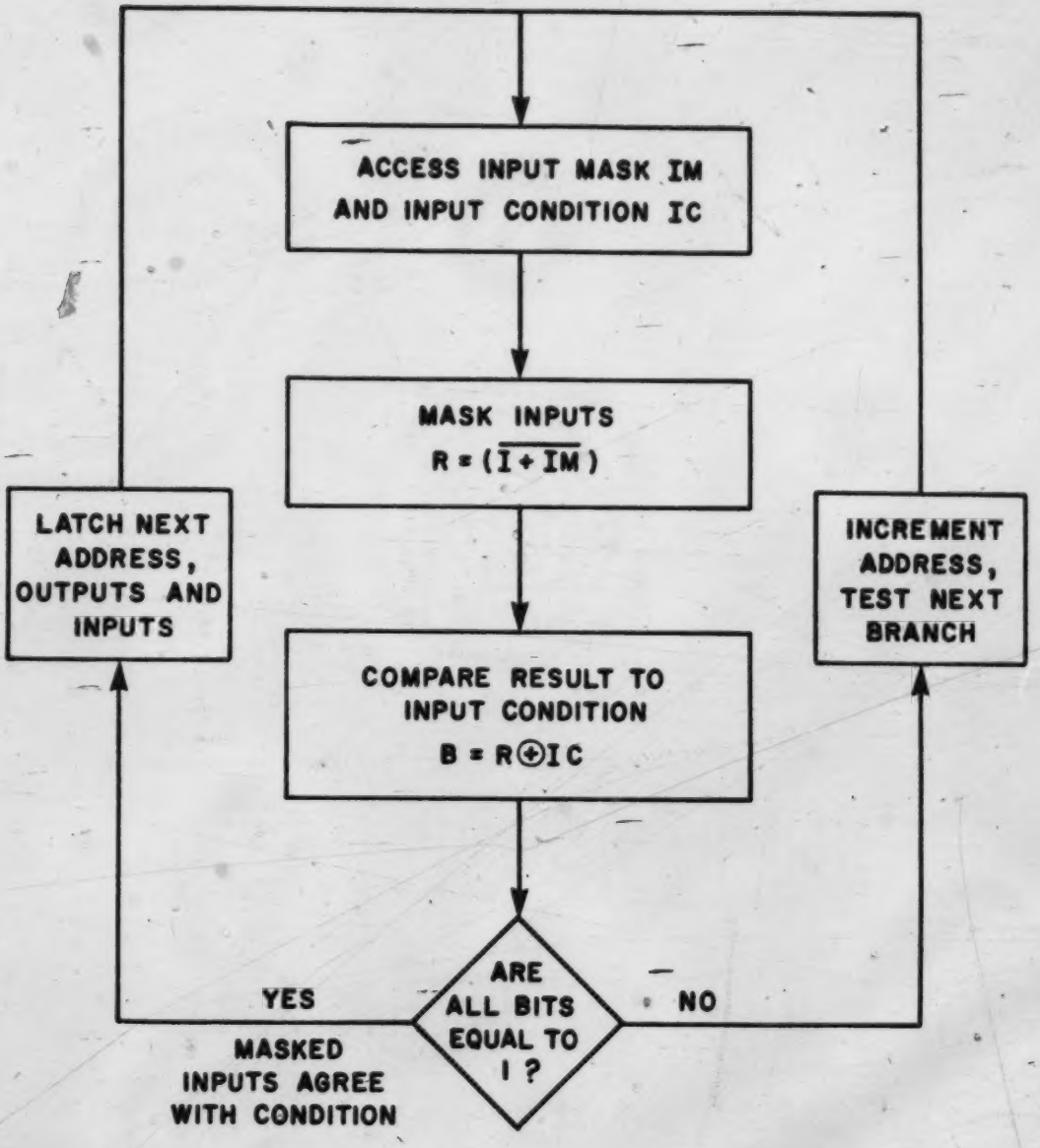


Figure 6. Memory segmentation flowchart.

STATE ADDRESS	INPUT MASK	INPUT COND	NEXT ADDRESS	OUTPUT	COMMENT
A: 000	01	11	010	01	BRANCH TO B
001	11	11	000	00	STAY AT A
B: 010	00	10	110	11	BRANCH TO C
011	00	11	010	01	STAY AT B
100	01	01	000	00	BRANCH TO A
101	11	11	010	00	STAY AT B
C: 110	00	11	000	00	BRANCH TO A
111	11	11	110	00	STAY AT C

Table 2. Memory segmentation microprogram for the FSM of Figure 1.

machine of Figure 1, using 32 three-bit words. For larger systems, memory segmentation clearly uses less PROM memory than the previous design. In Table 2, the input mask is derived by setting the input "don't cares" to 1 and the "cares" to 0. The input condition sets "don't cares" to 1 and leaves "cares" as is. The next address is the PROM address of the destination state. The outputs remain as is, with "don't cares" set arbitrarily to zero.

The memory segmentation algorithm may be used in the design of a PSC. Figure 7 shows a PSC designed to process the four data fields in serial. The fields would be stored on the same PROM, so a two-bit counter for the two lowest address bits would be used to distinguish between data fields (Table 3.a). On the first clock cycle, with the counter at 00, the input is latched and NORed to the input mask accessed from PROM. On the second clock cycle with the counter at 01, the result is EXORed with the input condition from PROM, and all bits are NANDed together to produce the $\overline{\text{LOAD}}$ signal. If the input matches the condition, then $\overline{\text{LOAD}}$ will be 0; on the third cycle, with the counter at 10, the output word is latched, and the new state address is latched on the fourth cycle with the counter at 11. Then the cycle continues, with the counter returning to 00 in the new state. If the input does not match the condition, then $\overline{\text{LOAD}}$ is 1, and the counter returns to 00 to begin a new cycle. At the same time, the present address is incremented to attempt the next highest priority branch.

The priority designated to a branch determines the clock cycles needed to to execute the branch. An n-th priority branch will take place in $2n+2$ clock cycles, since two clock cycles are required for each unsuccessful branch, and four for the successful branch.

Another design, shown in Figure 8, processes all data fields in parallel. No counter is required since each field is accessed at the same time at the same

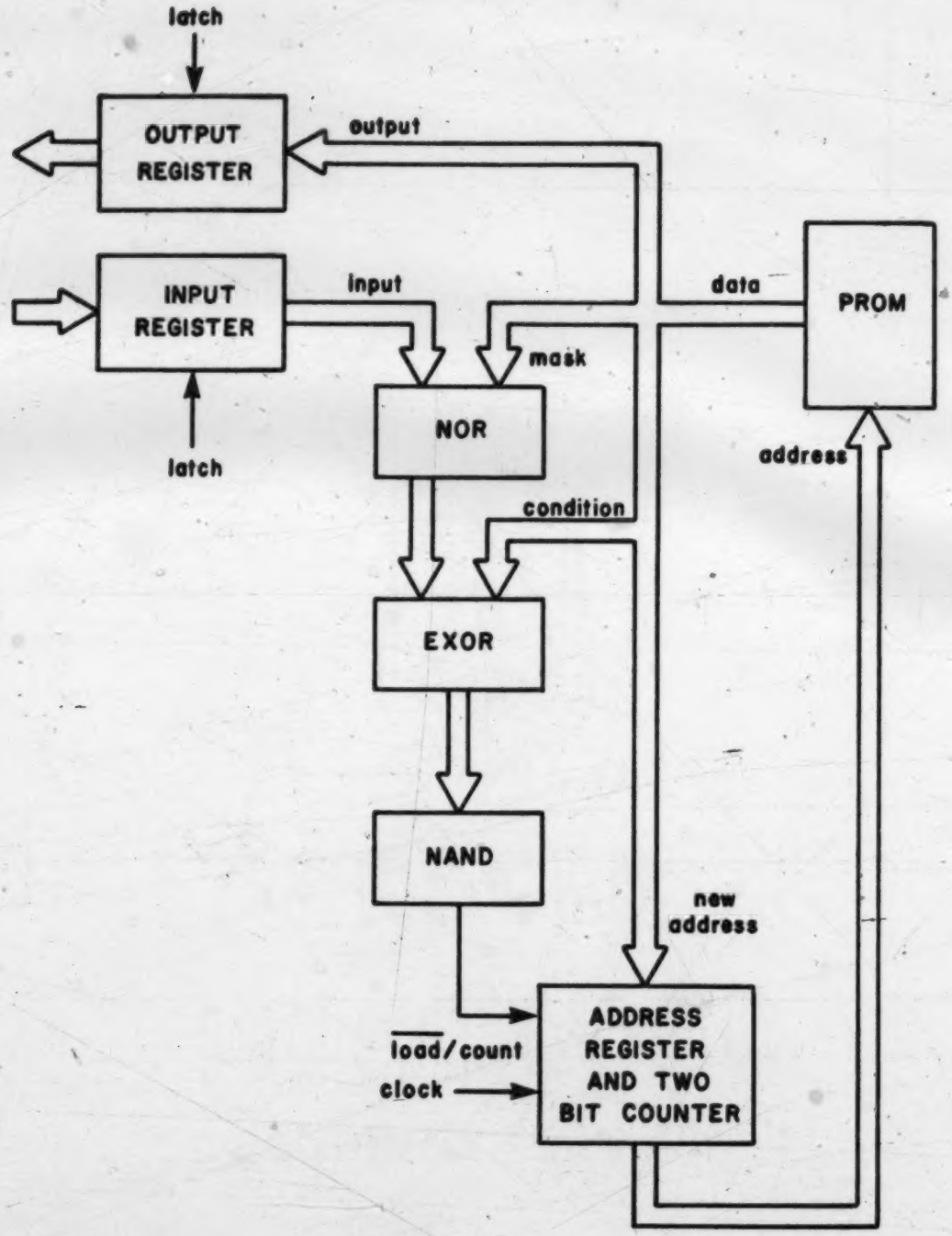


Figure 7. Controller design using serial memory segmentation.

COUNTER PRESENT	NEXT	CONDITION	COMMENT
00	01	XX	'NOR' INPUT AND MASK
01	00	BAD INPUT	'EXOR' WITH CONDITION AND 'NAND' ALL BITS TOGETHER
	10	VALID INPUT	
10	11	XX	LATCH_OUTPUT
11	00	XX	LATCH NEW ADDRESS

a)

COUNTER PRESENT	NEXT	CONDITION	COMMENT
0	0	BAD INPUT	'NOR' INPUT WITH MASK, 'EXOR' WITH INPUT COND. AND 'NAND' ALL BITS
	1	VALID INPUT	
1	0	XX	LATCH OUTPUT AND ADDRESS

b)

XX -- Transition occurs on all conditions.

Table 3. Transition tables for lowest address bits for a) serial and and b) semiparallel memory segmentation designs.

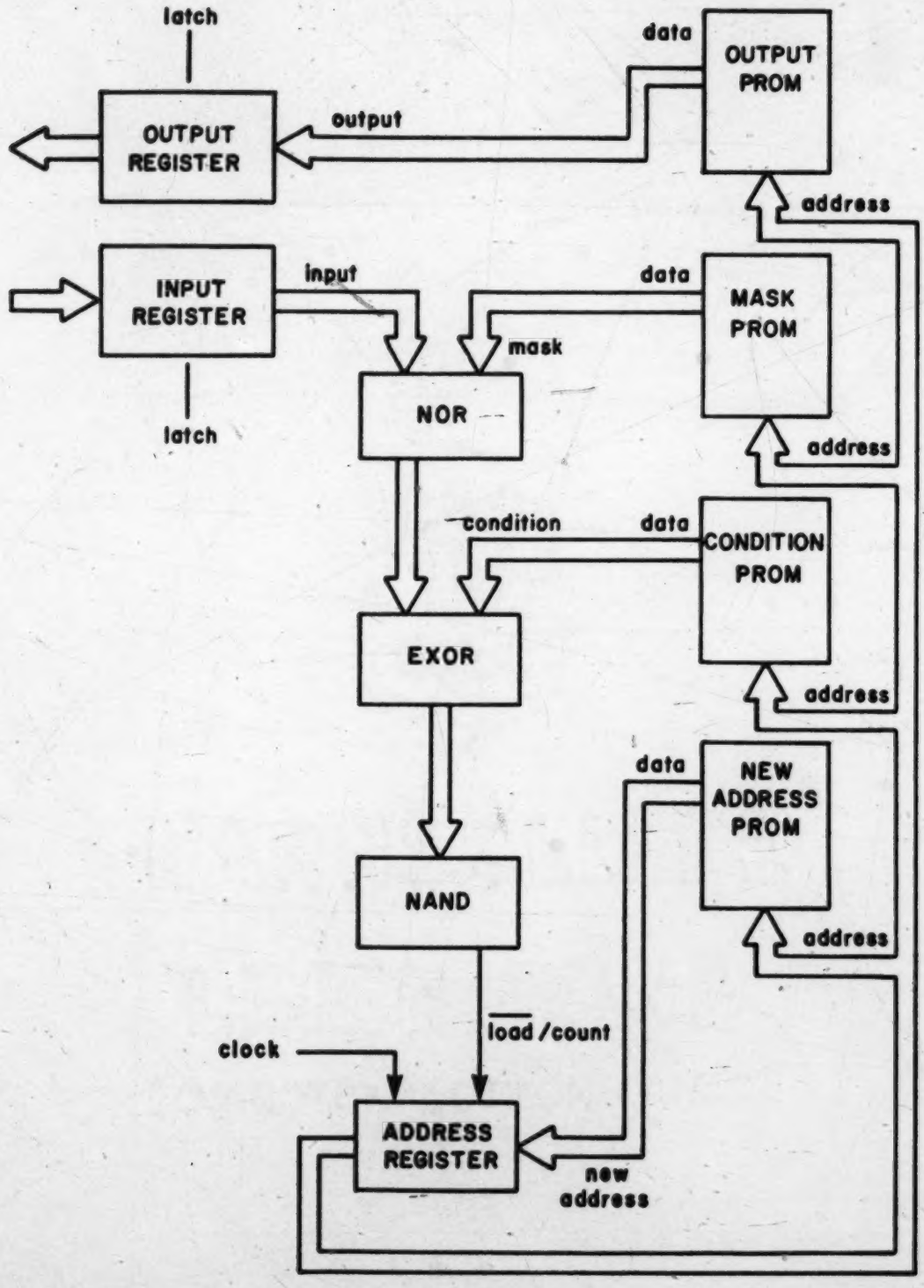


Figure 8. Controller design using parallel memory segmentation.

address on four different PROMs. During the same clock cycle, the input is NORed with the mask, EXORed with the condition, and all bits NANDed together to produce $\overline{\text{LOAD}}$. If it is high, the address is incremented to test the next highest priority branch. If it is low, the output and new state address are latched.

An n-th priority branch will require n clock cycles in the parallel PSC configuration. This is much faster than the serial PSC: four times as fast for the highest priority branch, and at least twice as fast for all branches. However, four separate PROMs are required for the parallel PSC, instead of just one. This decreases the ease of programming and installing the system.

A third design, shown in Figure 9, incorporates characteristics of both of the previous designs. The data fields are of two types: the input mask and condition are logic-related; the output and new address are output related. Therefore it seems reasonable to consider a semiparallel process which accesses the logic-related fields on the first clock cycle and the output-related fields on the second. This would allow the mask and output to be stored in order on one PROM, and the condition and new address to be stored on another. A one-bit counter would distinguish between logic and output fields, as shown in Table 3.b.

On the first clock cycle, with the counter (low address bit) at 0, the mask and condition are accessed from the two PROMs, and $\overline{\text{LOAD}}$ is determined. If it is high, the counter remains at 0 on the second clock cycle, and the next branch is tested. If it is low, the counter is 1 on the second cycle, and the new address and output are loaded. The counter returns to 0 on the next cycle in the new state.

An n-th priority branch will require n cycles to access the logic fields and one cycle to access the output fields, for a total of n+1 clock cycles. This is almost as fast as the parallel PSC design (one clock cycle slower), while

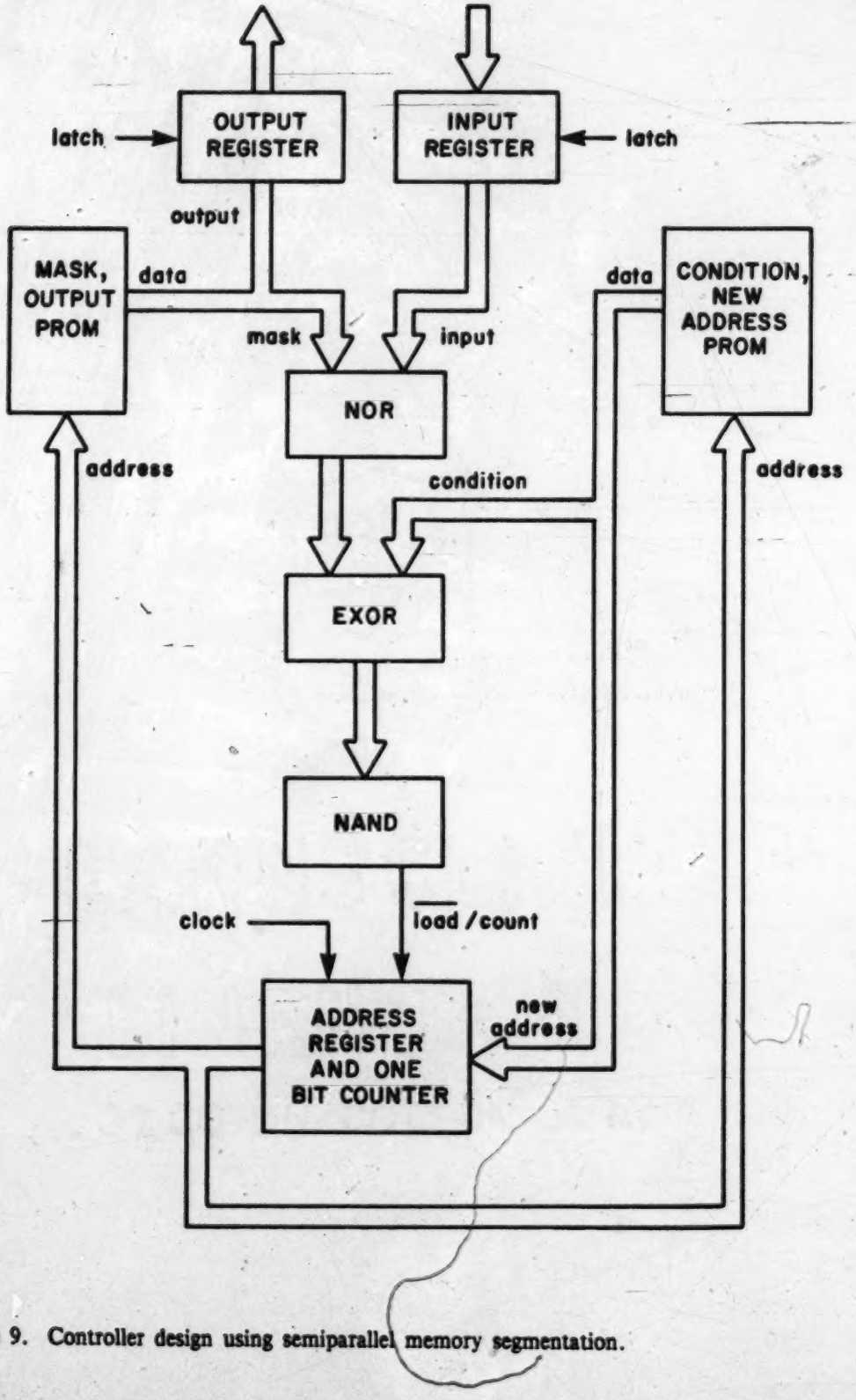


Figure 9. Controller design using semiparallel memory segmentation.

using only two PROMs instead of four.

Besides providing speed, memory efficiency and programmability, memory segmentation techniques can result in very powerful controllers with the addition of a control PROM and control lines to permit subroutines, interrupts, and input/output control.

Subroutine calls may be added by dividing PROM memory into equal parts as determined by the high address bit. The lower half is the main program memory; the top half is the subroutine memory. In the main program, three data fields are needed to define a subroutine call: subroutine address, return address, and subroutine-call command. The call subroutine command would be a single bit on the control PROM word, while the other fields would be stored on the regular PROMs.

Upon receiving the subroutine-call command, the PSC stores the return address in a special register and loads the subroutine address into the present address register, while setting the highest address bit high. When the subroutine program is completed, the same subroutine-call bit forces the highest address bit low again and causes the return address to be loaded in the present address register. Execution of the main program then continues.

In this way, for a word length of n bits, there will be 2^n words available for the main program and 2^n words for any number of distinct subroutines. However, nesting of subroutines will not be possible without a stacked register to hold return addresses.

Interrupts may be handled in a similar manner. Upon receipt of an interrupt (and subsequent acknowledgment) via control lines, the present address is stored in the return address register. Then, the present address register is loaded with the address of the top word of memory. At this location, an

unconditional jump to the interrupt service routine takes place. Upon completion of the service routine, the return to the main program is identical to that for a subroutine.

A bit in the control word on the control PROM may be specified as an interrupt mask connected to the PSC via a control line. When the bit is low, interrupts can occur; when high, interrupts are ignored. This is necessary, as interrupts must be prevented in some instances; for example, during the servicing of a subroutine or another interrupt.

The remaining bits on the control word may be used for control of peripherals such as input/output ports, input/output modes, and programmable timers. The result will be a very powerful and versatile programmable sequential controller.

II INTEGRATED CIRCUIT DESIGN

2.0 Introduction

A PSC designed using the memory segmentation technique would require a considerable amount of circuitry in the form of logic gates (NAND, NOR, EXOR), storage registers, and multiplexers. The PSC could be constructed from medium scale integrated (MSI) devices, but such a circuit would be fairly large and cumbersome, especially for larger word lengths.

Instead, it is desirable to implement the entire PSC circuit on a single silicon chip using large scale integration (LSI) techniques. Such a chip could be easily installed into a larger system; only a pin-out diagram and a basic understanding of the chip's operation would be needed. The integrated circuit (IC) would be subject to fewer wiring problems such as crossed or broken wires than the MSI circuit. Marketability of the PSC would also be tremendously enhanced.

The IC was designed using digital design techniques and was layed out with computer-aided graphics. The circuit followed the Northern Telecom CMOS 1B rules [40] and was fabricated by the Canadian Microelectronics Corporation (CMC) in Kingston, Ontario, Canada.

2.1 Design specifications

An integrated circuit is to be designed to implement a programmable sequential controller using memory segmentation techniques to provide greater speed, memory efficiency, and programmability.

The IC (or a number of cascaded ICs) should have a data stream width of at least eight bits. Obviously, an unlimited width data stream would be ideal to accomodate any industrial control application, but an eight-bit word may be enough if the control PROM word is used to select input/output ports. It should be noted that PROMs typically have an eight-bit word length. However, this can be expanded by wiring two or more PROMs in parallel.

The chip should be able to handle interrupts and allow the masking of interrupts via a control bit. An interrupt should send control to a specified service routine. At the end of the routine, control should return to the address where the interrupt occurred.

The chip should also handle subroutines, with the address of the subroutine and the return address specified in program memory at the point of the subroutine call.

The IC should have a control line to reset the PSC to its initial state.

2.2 System architecture

The simplest way to provide an eight bit wide data stream is to design an eight-bit wide circuit. Table 4 shows that such a design would require a large number of pins or lines per chip (including address, data, and control lines): 43 pins for the serial memory segmentation design, 50 for the semiparallel, and 65 for the parallel. The inability of the eight-bit design to handle a wider data stream is another disadvantage.

An alternative would be to design a four-bit IC that could be cascaded to to produce data streams of any multiple of four bits. From Table 4, this design would require 27, 30 or 37 pins, as well as additional control lines to make the chip cascadable.

In all, six additional control lines would be necessary to support cascadability. In order to increment the present address during a branch test, a carry-in must be added to the lowest address bit. Since this must only be done on one chip (the chip handling the least significant address bits), the connection cannot be made internally. Instead, a carry-nought (Co) line must be designed. Also, each chip must have carry-in (CI) and carry-out (CO) lines to permit the address-increment to ripple through the PSC. The Co of the lowest chip would be connected to the CI of the same chip; its CO pin would connect to the CI of the next chip, and so on. The CO signal of the highest chip could be used as an overflow signal.

In order to determine if the load of a new address should occur, the signal $\overline{\text{LOAD}}$ must be produced. This may be done by calculating the $\overline{\text{LOAD}}$ for each chip, and ORing the signals together. This is so because if

$$\overline{L1} = \overline{A\,B\,C\,D},$$

$$\overline{L2} = \overline{E\,F\,G\,H},$$

and

DESIGN	INPUT	OUTPUT	ADDR	LOW	HIGH	DATA	RESET	CLOCK	INT	SUB	POWER	GROUND
SERIAL	n	n	n	2	1	n	1	1	3	1	1	1
SEMIPAR.	n	n	n	1	1	2n	1	1	3	1	1	1
PARALLEL	n	n	n	0	1	4n	1	1	3	1	1	1

n -- refers to the data stream width.

LOW -- refers to the low address bit(s) needed to distinguish between data fields.

HIGH -- refers to the high address bit distinguishing the main program from the subroutines.

INT -- refers to the external interrrupt, interrupt acknowledge, and interrupt mask control lines.

SUB -- refers to the call subroutine control line.

TOTAL PINS:

SERIAL: 4n + 11

SEMIPARALLEL: 5n + 10

PARALLEL: 7n + 9

Table 4. Number of pins for various IC configurations.

$$\overline{L3} = \overline{I\ J\ K\ M}$$

are the load signals produced by three chips, then the load signal for the entire system will be

$$\overline{LOAD} = \overline{A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ M}$$

$$= \overline{L1\ L2\ L3}$$

$$= \overline{L1} + \overline{L2} + \overline{L3}\ .$$

So, in order to cascade the chips, the load signal of one chip must be ORed with the signal of the next chip, and this result ORed with the signal of the following chip. Therefore, two lines are needed, designated NO (NAND-out) and NI (NAND-in), with NO being the load signal produced by a chip, and NI being the input to the OR gate from the previous chip. This means that the NI of the lowest chip is set to zero, while the NO of the highest chip is the result, $\overline{LOAD}$, which must be fed into each chip through another control line ($\bar{L}$).

Figure 10 shows how three ICs using the six control lines would be cascaded. This would provide a twelve bit word length and allow 4K (4096) main program branch instructions.

The maximum number of chips that can be cascaded together will clearly depend on the propagation delays of the various signals, and on the clock frequency. For example, the load signal must propagate through the chips and be present at the $\bar{L}$ input of each chip in less than one clock cycle. This provides an upper limit on the number of chips in the system.

Table 5 compares the characteristics of the various memory-segmented PSC designs. Clearly the cascadable design is preferable because it allows an almost unlimited data word width and program size, while having a reasonable number of pins per chip. The semiparallel design is the optimum: its branching speed is almost as good as that of the parallel design (n+1 versus n cycles), while the

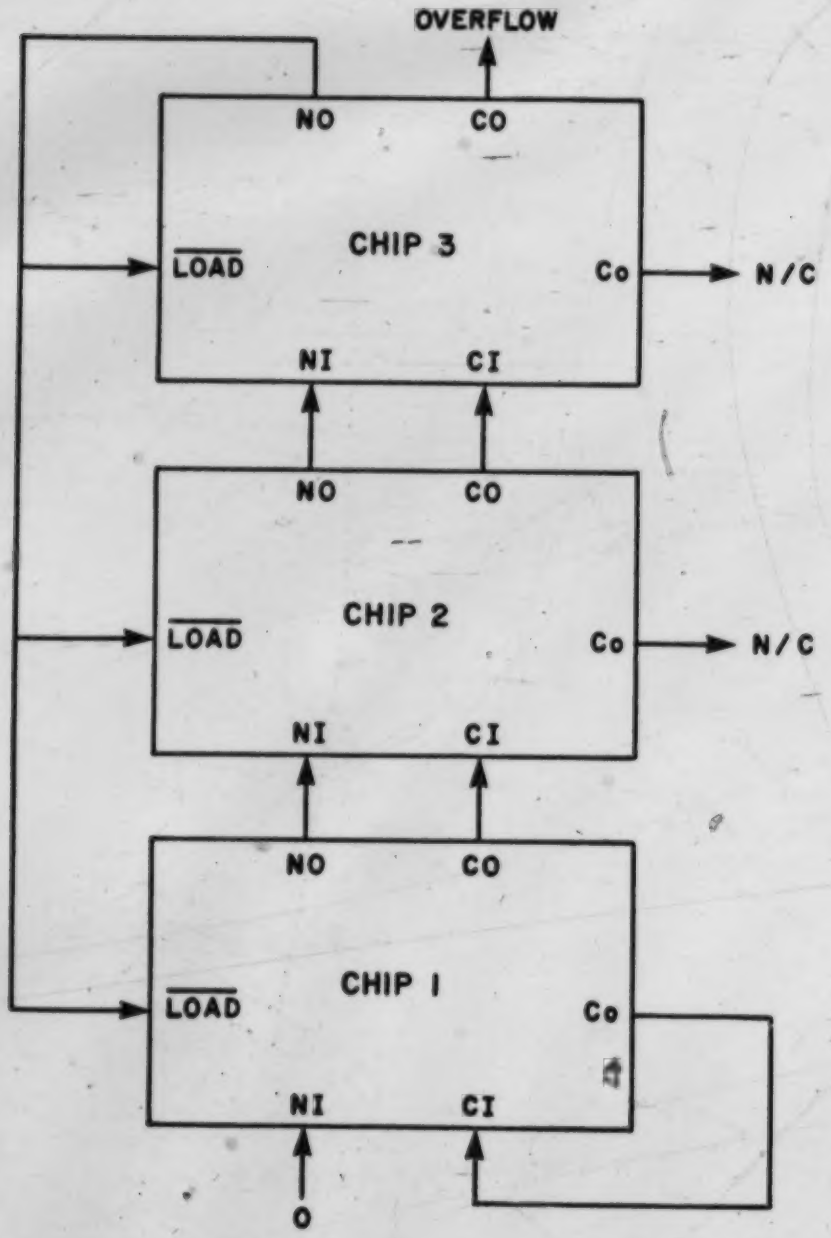


Figure 10. Block diagram of cascaded PSC chips.

CONFIGURATION	MAX # OF BITS	PINS PER CHIP	PINS FOR 8 BITS	CYCLES PER BRANCH**	MIN # OF PROMS
8 BIT SERIAL	8	43	43	2n + 2	2
4 BIT SERIAL CASC.	*	33	66	2n + 2	2
8 BIT SEMIPARALLEL	8	50	50	n + 1	3
4 BIT SEMI. CASC.	*	36	72	n + 1	3
8 BIT PARALLEL	8	65	65	n	5
4 BIT PAR. CASC.	*	43	86	n	5

* -- The maximum number of bits for a cascadable system will depend on how many chips can be cascaded, a function of the clock frequency and the chips' internal delay times.

** -- The number of clock cycles per branch is calculated for an n-th priority branch.

Table 5. Comparison of different IC configuration characteristics.

complexity of installation and operation is almost as low as the serial design (three versus two PROMs per PSC; 36 versus 33 pins per chip).

Thus, the IC will be designed to implement the cascadable semiparallel techniques previously discussed. The proposed IC will have a four bit wide data path, 36 pins, an nth priority branch time of n+1 cycles, and two external data PROMs and a control PROM.

2.3 Floor plan

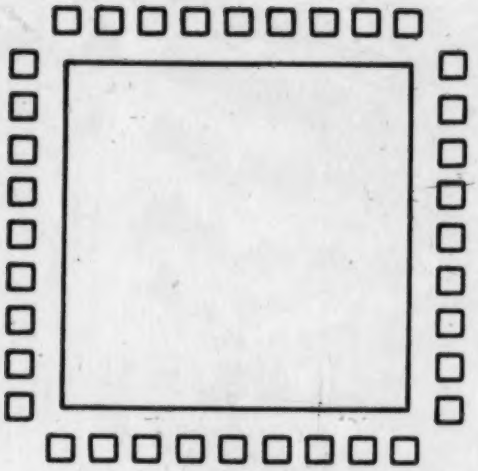
The optimum integrated circuit design will be square in dimension, use the smallest amount of silicon area, and minimise bus routing within the circuit. In order to achieve such a design, a series of floor plans is constructed to break the circuit up into successively smaller functional blocks which use inputs from previous blocks and produce outputs for other blocks (much in the way a program is written by breaking it up into subroutines or procedures). The smaller blocks may then be positioned so as to minimise wasted area and place the outputs of one block near the corresponding inputs of another block.

Figure 11.a shows the initial floor plan: a square with 36 inputs and outputs specified on its perimeter. This square will consist of three blocks, as shown in Figure 11.b: the logic block handles the production of the load signal; the address block handles the latching of the new state address; and the control block produces the circuit control signals such as the one-bit counter, the latch clocks, and the subroutine control bits. Since the address block will be the largest, it becomes the bottom half of the square IC, while the other two blocks compose the upper half.

The components of the logic block are shown in Figure 12.a. They consist of input/output registers, a two-input NOR section to mask irrelevant inputs, a two-input EXOR section to check the input condition, and a quadruple NAND gate and two-input NOR gate to calculate the NO signal. The external inputs are the input, output, mask, and condition words and the NI signal. The external outputs are the latched output word and the NO signal.

Each section will have four gates in parallel to accommodate a four-bit word length. These will be placed one atop the other. Thus, it is desired to place all sections of the logic block on one level, as shown in Figure 12.b.

a)



b)

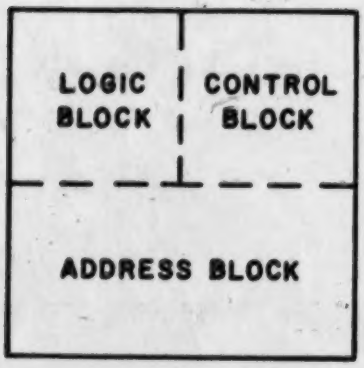


Figure 11. Initial floor plan.

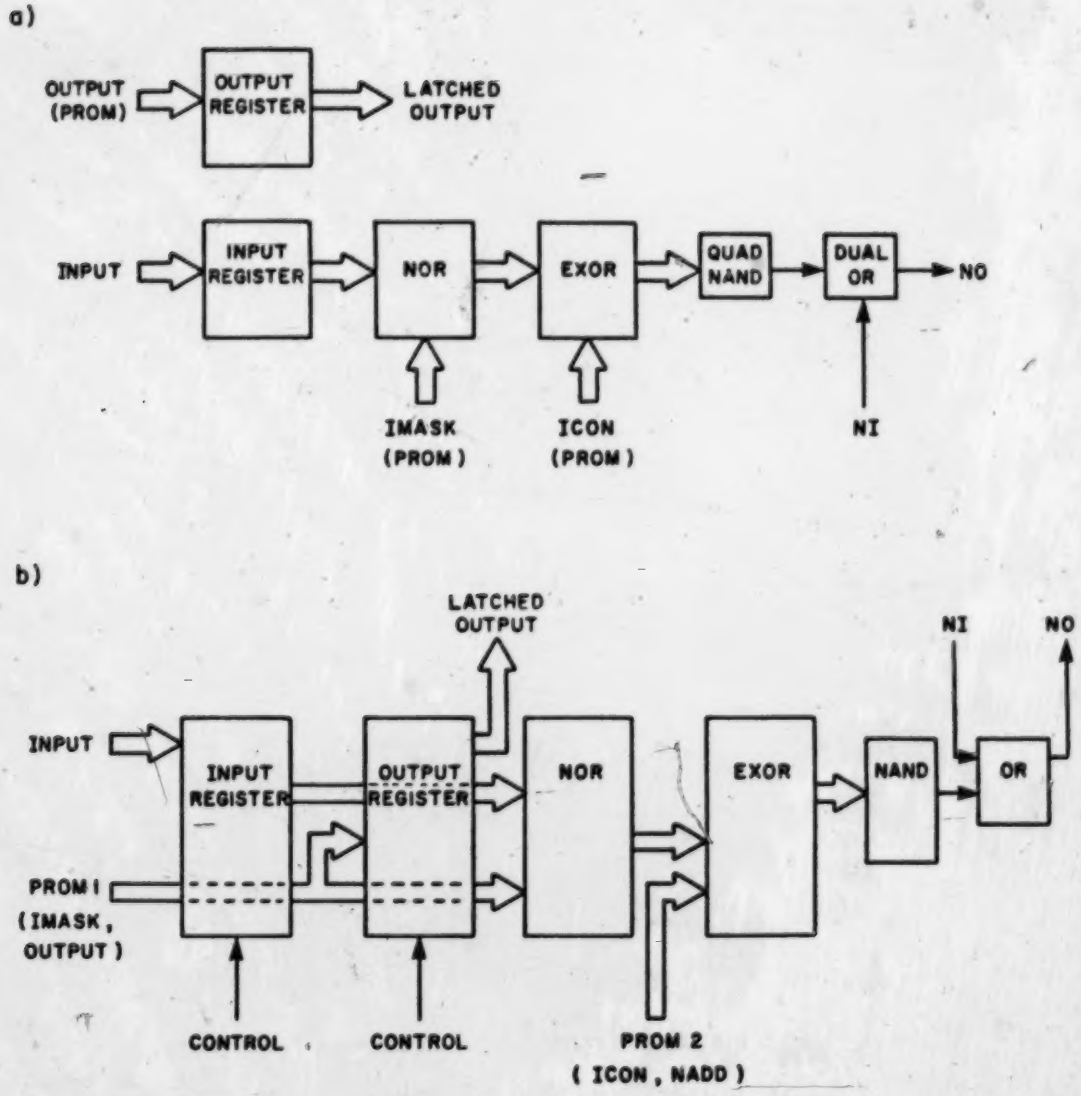


Figure 12.—Logic block floor plan.

This will produce the most compact design and minimise the amount of routing across levels. Since the output register has been placed near the input mask word, the mask and output will be stored on the same PROM (PROM1) and will be distinguishable only by the one-bit counter C1, which represents the lowest address bit on the PROM. The input condition and new address will be stored on the other PROM (PROM2).

The external inputs and outputs of the logic block are situated at the edges of the IC, where they will be connected to input and output pads.

The components of the address block are shown in Figure 13.a. They are: a multiplexer (MUX2) which selects either the present address (in case of interrupts) or the PROM-stored address (for subroutines) as the return address; a return address register; a multiplexer (MUX5) to select either the reset address 0000, the interrupt address 1111, the incremented address, the return address, or the new address; a half-adder section to increment the present address; and a present address register. The external inputs are the PROM-stored return address and new address, and the carry-in (CI). The external outputs are the present address and the carry-out (CO).

Because the reset address is just the interrupt address incremented by one, the same signal may be used for both reset and interrupt, with the carry-in to the half-adder either set or cleared. To accomplish this, the half-adder is placed following the multiplexer MUX5, with CI set high only for address increment and reset.

It is desirable to place all sections of the block on the same level. This will minimise routing across levels (the present address is required in several sections of the block and must be routed there from the present address register). This will also make the design more compact: the block will be four bits high and

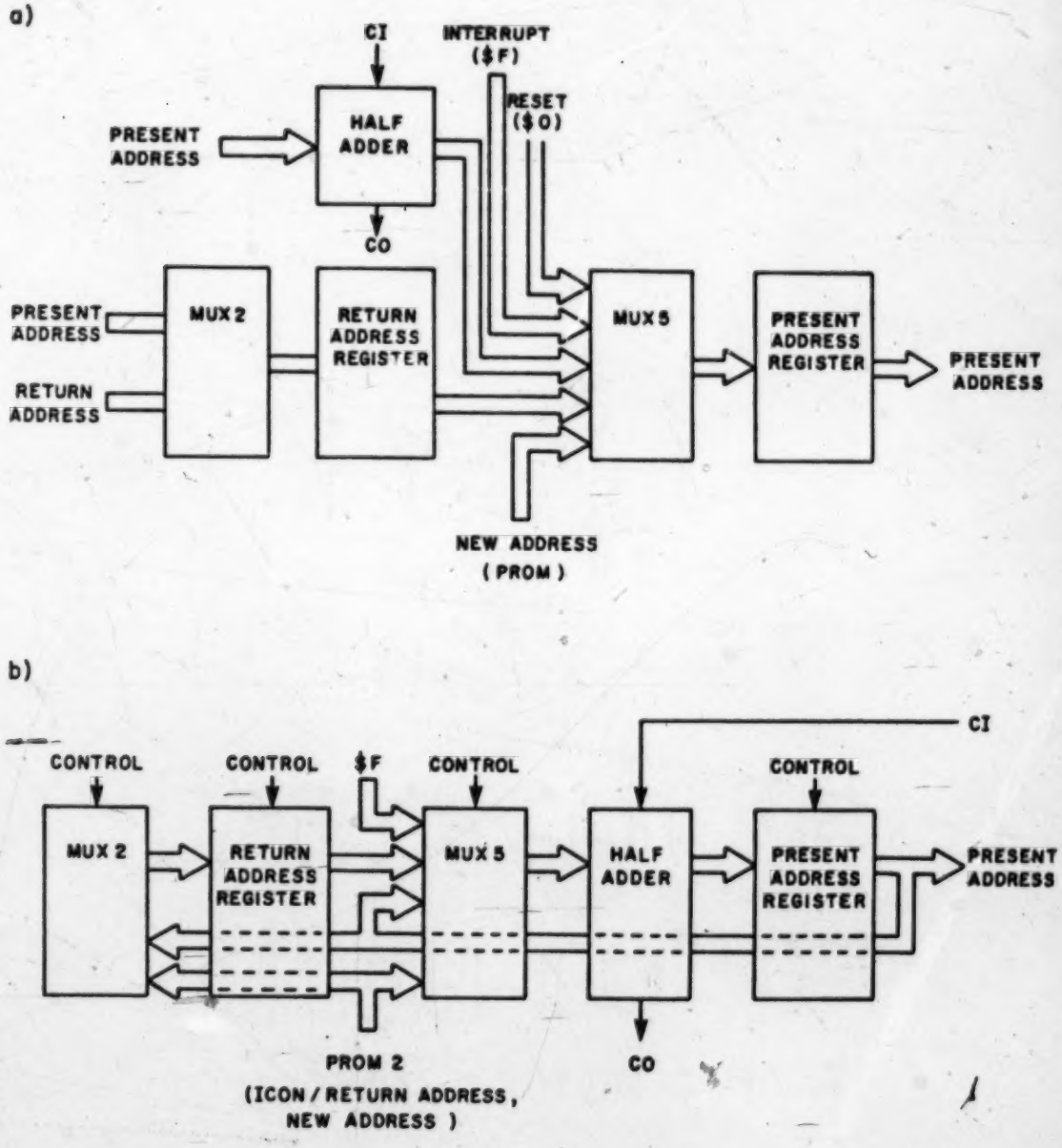


Figure 13. Address block floor plan.

five sections long, forming the bottom half of a fairly square IC. Figure 13.b shows the proposed floor plan for this block. Because the MUX2 is near the MUX5, the return address should be stored on the same PROM as the new address. This means that the return address will replace the input condition during a subroutine call.

The external inputs and outputs are again situated along the edges of the IC, to be connected to I/O pads.

The control block, as shown in Figure 14.a, consists of the one-bit counter C1, the subroutine address bit SB, and the logic which controls interrupts, initial carry-in, and input/output/address register latches. External inputs include load signal $\bar{L}$, interrupt signal EINT, interrupt mask IM, reset signal RES, clock CK, and subroutine call bit CS. The external outputs are the low address bit C1, the high address bit SB, the interrupt acknowledge signal IACK, and the initial carry-in Co.

The low address bit C1 and the high address bit SB will consist of logic to compute the value of C1 and SB on the next clock pulse, and registers to latch these values. These sections will be many gates long and one gate high, so they are place one atop the other in the upper right-hand block of the IC. The interrupt and carry-in logic is located on the next higher level, while the register latch signal logic is placed as close as possible to the corresponding register to minimise routing. This is accomplished by introducing a level of gates in the centre of the IC, to separate the logic and control blocks from the address block and allow the latch signals to have easy access to the registers. The modified control block floor plan is shown in Figure 14.b.

Figure 15.a shows the floor plan for the whole integrated circuit. This plan will be used only as a guide to the layout of the actual gates; some

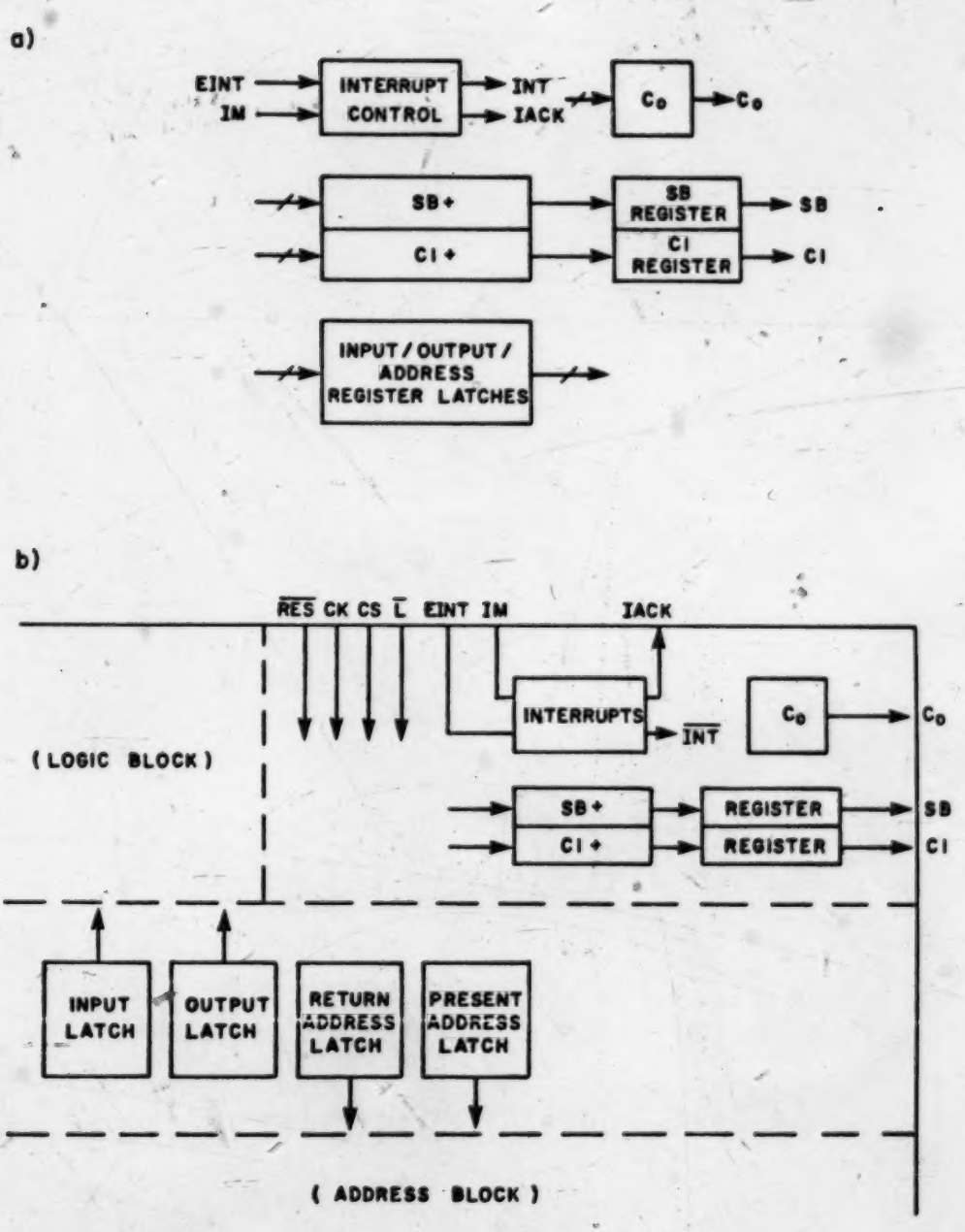
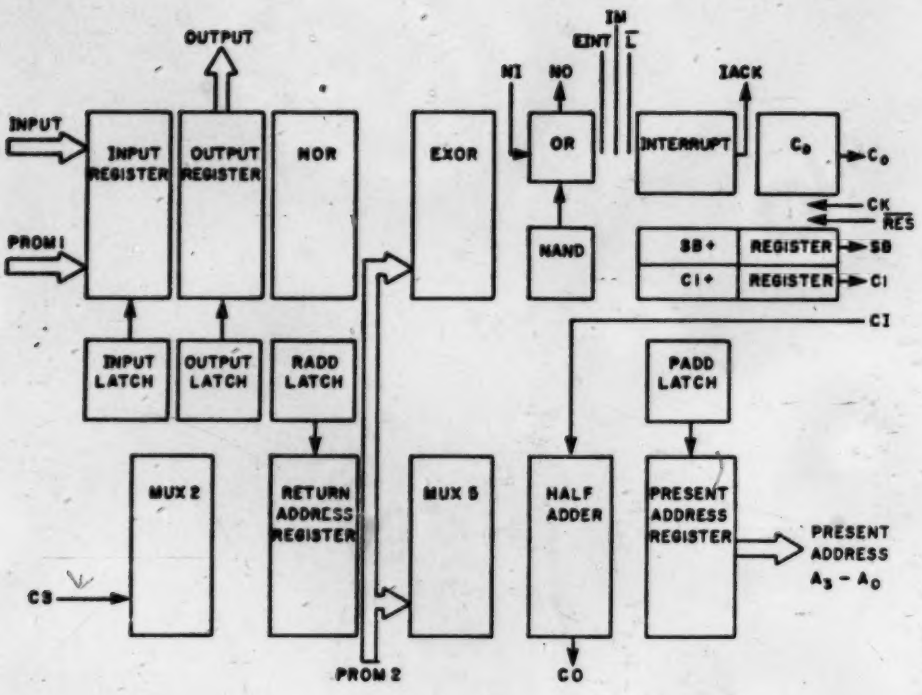


Figure 14. Control block floor plan.



IN 0	OUT 0	NOR 0	XOR 0	OR		
IN 1	OUT 1	NOR 1	XOR 1	NAND 4	INT	C_0
IN 2	OUT 2	NOR 2	XOR 2		SB+	SB REG
IN 3	OUT 3	NOR 3	XOR 3		CI+	CI REG
INCK	OUT CK	RADD CK				NADD CK
MUX2 0	RADD 0	MUX 5 0	ADDER 0			NADD 0
MUX2 1	RADD 1	MUX 5 1	ADDER 1			NADD 1
MUX2 2	RADD 2	MUX 5 2	ADDER 2			NADD 2
MUX2 3	RADD 3	MUX 5 3	ADDER 3			NADD 3
V_{DD}						GND

Figure 15. Final floor plan.

modifications may be made in order to achieve a more compact design or fewer routing conflicts.

Figure 15.b shows the floor plan in gate levels with the power-ground net inserted. This net is a set of interdigitated combs [29] which allows the power and ground lines to reach all parts of the IC without crossing each other. Power and ground signals are available at opposite ends of each gate; two adjacent levels share either the same power line or the same ground line. This means that every second gate level will be "upside-down" with respect to the adjacent gate level. The power and ground lines encircling the IC will provide power and ground to the input protection pads and the output driver pads, and form the "pad frame" of the circuit.

2.4 Logic design

The next subdivision in the design process involves the logic design of each section of the circuit. This will involve the evaluation, minimisation, and implementation of the functions describing each section. The function is evaluated by translating the description of operation of the circuit section into a truth table, detailing the state of the function for every possible input. Karnaugh maps [23], which eliminate "don't care" inputs and reduce terms by grouping adjacent minterms, can then be used to minimise the function. Finally the function is expressed in terms of basic building-block gates (NAND, NOR, EXOR). These basic gates will later be designed at the transistor level using complementary metallic oxide semiconductor (CMOS) field effect transistors (FETs).

The functions of the logic block of the IC have already been determined and need only be broken down to the gate level. The logic block has been described as a four-bit block of input/output registers, two-input NOR gates, and two-input EXOR gates, and a four-input NAND gate and an OR gate, as shown in Figure 12. Only the registers and the OR gate need to be broken down further.

The data register function may be implemented using a clocked D-type flip-flop, shown in Figure 16.a. This circuit allows the data bit D to propagate to the output Q while the clock is high and holds this signal while the clock is low. The circuit, however, uses a large number of gates and is not practical for LSI design.

Figure 16.b shows an alternate implementation of the register function [29]. It uses transmission gates to control the flow of the signal and inverters to store the signal. The operation of this circuit is quite simple. When the clock is high the first transmission gate opens and the second closes; the data is present at the inverters and at the output. When the clock is low, the first gate

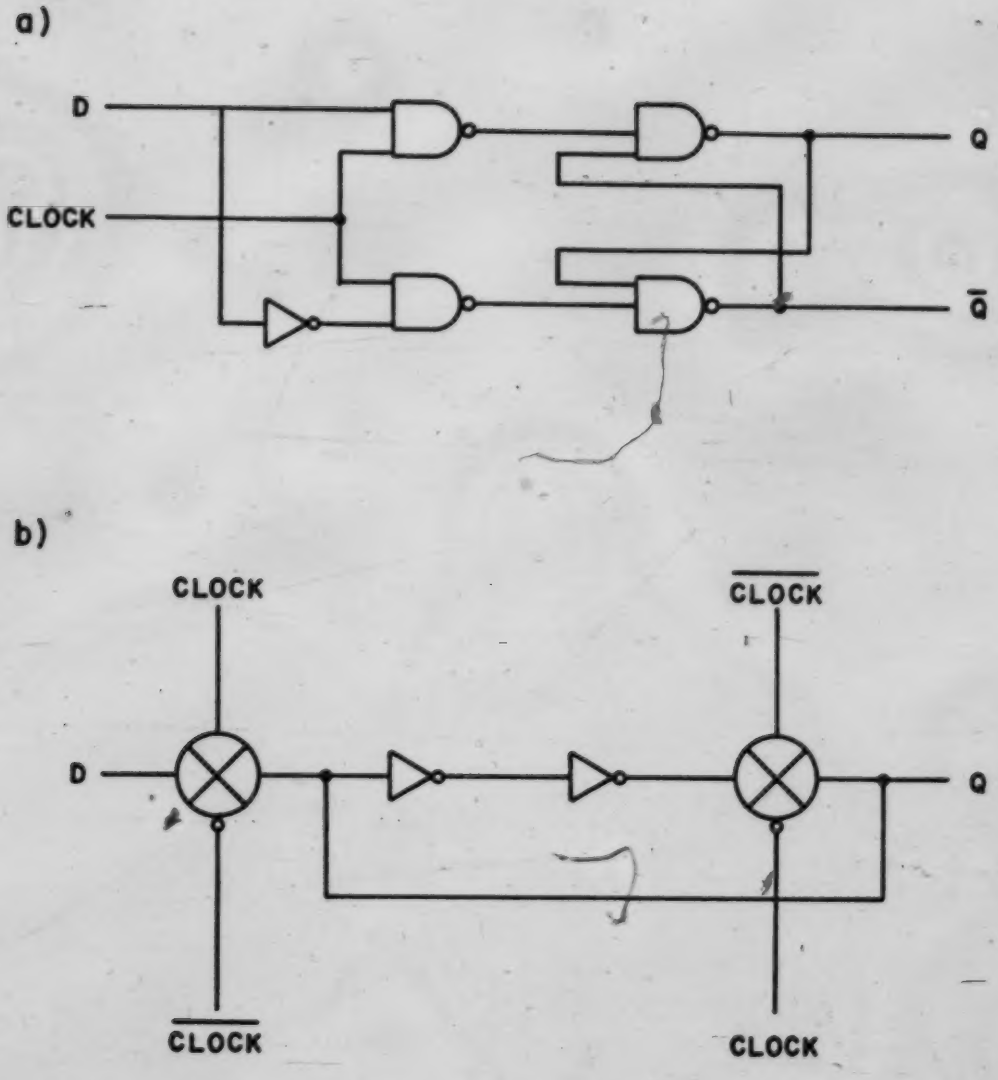


Figure 16. Gate diagrams of data registers using D flip-flop and transmission gates.

closes and the second opens, allowing the data to be held by the feedback through the two inverters. The transmission gate register will be much more compact than the flip-flop register since a transmission gate is simply two complementary transistors in parallel.

These registers are level-triggered. This means that if the input D changes while the clock is high, the output Q will change as well. In cases in which the input is a function of the output, edge-triggered registers which latch only on the rise or fall of the clock pulse will be needed to produce one output latch per clock cycle. Figure 17 shows two methods of producing an edge-triggered register. In Figure 17.a the clock pulse is ANDed with its inverse signal. This results in a clock pulse which will be high for only one inverter delay, while the clock is high and its inverse signal has not yet been propagated. A more reliable method (which does not depend on the length of an inverter delay or on the amount of time that the clock must be high in order to latch the data) is the master-slave register shown in Figure 17.b. The master register passes data on the clock high while the slave passes on the clock low. The result is that data can be passed only once per clock cycle.

The input and output registers of the logic block need not be edge-triggered, since their inputs are not a function of their outputs. The input and output latch clocks will be designed in the control block.

The OR gate is designed as a NOR gate with an inverter, since negative logic gates are much easier to design in CMOS than positive logic gates.

The address block consists of the MUX2, MUX5, half-adder, return address register, and present address register. The registers will be edge-triggered using the technique of Figure 17.b with latch clocks to be designed in the control block.

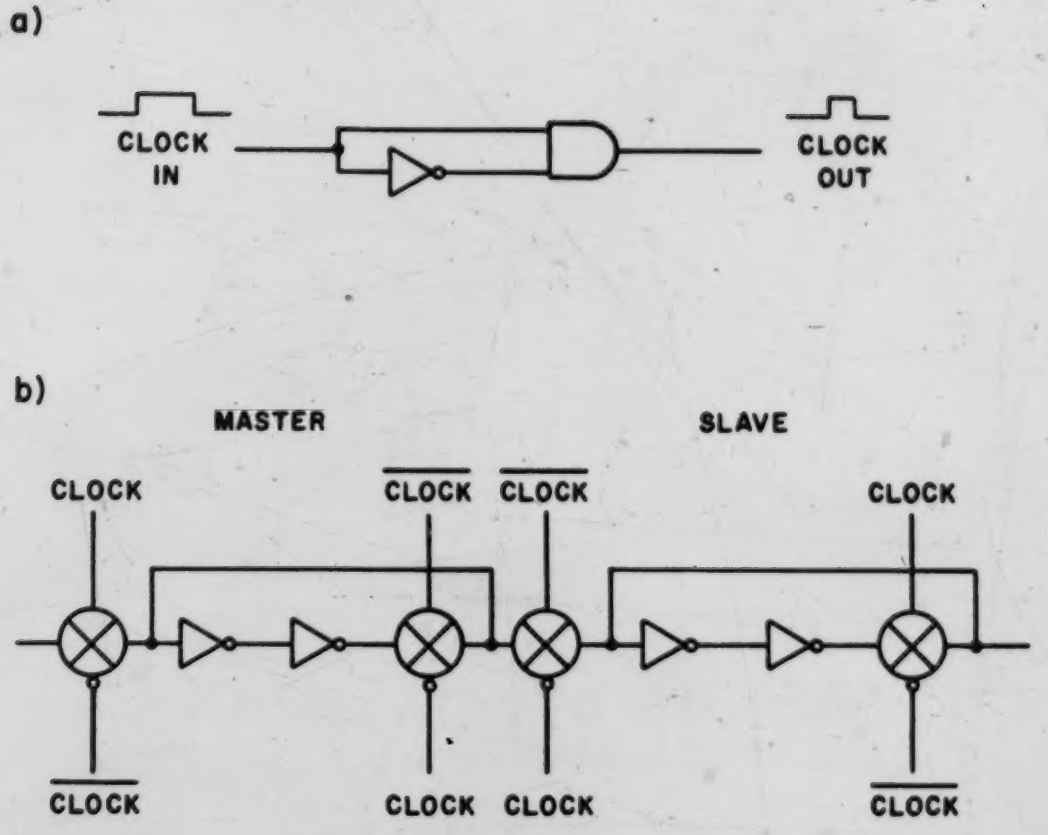


Figure 17. Edge-triggered register circuits.

The MUX2 must select either the return address on a subroutine call or the present address on an interrupt. Interrupts must be masked during subroutine calls since return addresses cannot be stacked; therefore, the call subroutine bit CS will be sufficient to select between present and return addresses. The MUX2 equation will be

MUX2 output = (CS)(return add) + ($\overline{CS}$)(present add)

in AND-OR representation. This equation is converted directly to NAND-NAND representation and is implemented as shown in Figure 18.a using three dual-input NAND gates.

The MUX5 must select among reset, interrupt, return address, new address, and present address. The truth table for the circuit is shown in Table 6.a. If reset signal $\overline{RES}$ (chosen active-low) is low, 1's are passed on all four bits (with the carry-in to the adder CI set high). If $\overline{RES}$ is high and internal interrupt signal $\overline{INT}$ (which has been previously masked or unmasked) is low, 1's are passed on all bits (with CI set low). With both $\overline{RES}$ and $\overline{INT}$ high: if both the one-bit counter C1 and bit CS are low, the present address is passed (with CI high), to increment the address; if C1 is high and CS is low, a branch is to occur, so the new address (PROM2) is passed; if C1 is high, CS is high, and the highest address bit SB is low, a subroutine is to be called, so the new address (PROM2) is again passed; finally, if C1, CS, and SB are high, a return from subroutine or service routine is required, so the return address is passed as the multiplexer output. Table 6.b shows the Karnaugh map used to minimise the MUX5 function by eliminating irrelevant input variables. Certain input combinations are listed as "don't cares" because the output due to those inputs is not of interest. The resulting MUX5 equation is

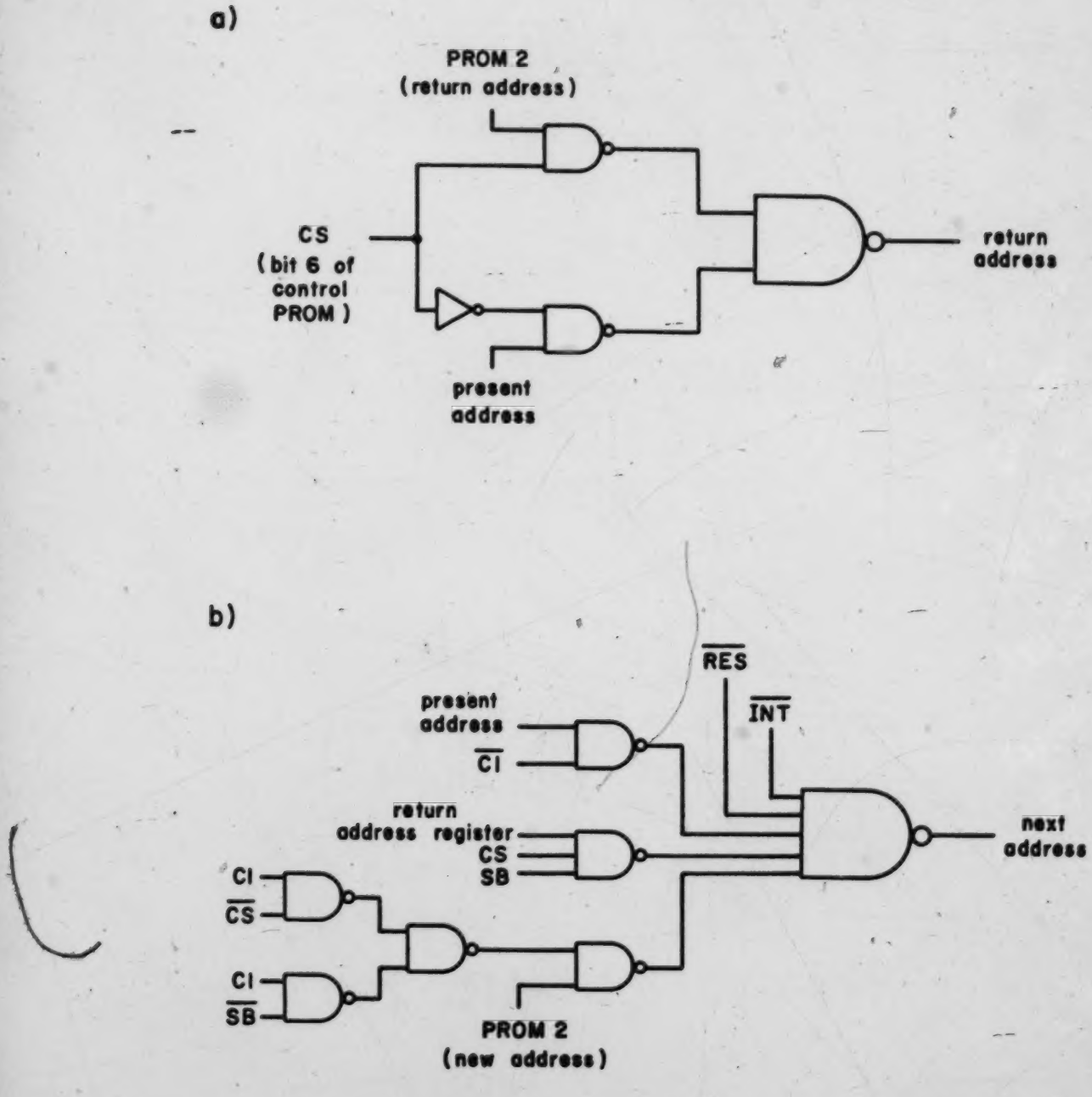


Figure 18. Gate diagrams of two-input and five input multiplexers.

$\overline{RES}$	$\overline{INT}$	C1	CS	SB	MUX5 OUTPUT
0	X	X	X	X	1
1	0	X	X	X	1
1	1	0	X	X	a (present address)
1	1	1	0	X	b (new address)
1	1	1	1	0	b (new address)
1	1	1	1	1	c (return address)

a)

$\overline{RES}$ $\overline{INT}$ \ C1 CS SB	000	001	011	010	110	111	101	100
00	1	1	1	1	1	1	1	1
01	1	1	1	1	1	1	1	1
11	a	a	X	X	b	c	b	b
10	1	1	1	1	1	1	1	1

b)

X -- don't care

Table 6. a) Truth table and b) Karnaugh map for five-input multiplexer.

$$\text{MUX5 output} = \overline{\overline{RES}} + \overline{\overline{INT}} + (\overline{RES})(\overline{INT})(\overline{C1})(\text{pres add})$$
$$+ (\overline{RES})(\overline{INT})(C1)(\overline{CS} + \overline{SB})(\text{new add})$$
$$+ (\overline{RES})(\overline{INT})(CS)(SB)(\text{ret add}) .$$

However, using the predicate calculus rule [36]

$$A + (\bar{A})(B) = A + B ,$$

the multiplexer equation simplifies to

$$\text{MUX5 output} = \overline{\overline{RES}} + \overline{\overline{INT}} + (\overline{C1})(\text{pres add})$$
$$+ (C1)(\overline{CS} + \overline{SB})(\text{new add}) + (CS)(SB)(\text{ret add}) .$$

The NAND gate implementation of this function, shown in Figure 18.b, uses a five-input, a three-input, and five two-input NAND gates.

The half-adder (incrementer) section will consist of a four-bit input, a carry-in CI, and a carry-out CO. It may be designed as a block of four half-adder cells, each producing the functions

$$S_i = A_i \oplus C_{in} \quad \text{and} \quad C_{out} = (A_i)(C_{in}) ,$$

with the C-out of one cell leading to the C-in of the next. The simplest circuit which produces both of these functions uses four two-input NAND gates and an inverter, as shown in Figure 19. The main consideration of the design should be the propagation delay of the carry. In this circuit, the carry delay will be two gates long per bit, or eight gates per IC. This delay should not be an important factor in the chip's operation.

The control block consists of the $\overline{INT}$, C1, SB, and Co functions, as well as the latch clocks for the input, output, return address, and present address registers.

The interrupt signal $\overline{INT}$ (active-low) will be designed to go low only when the EINT input is high, the IM (interrupt mask) input is high, $\overline{RES}$ is high, and

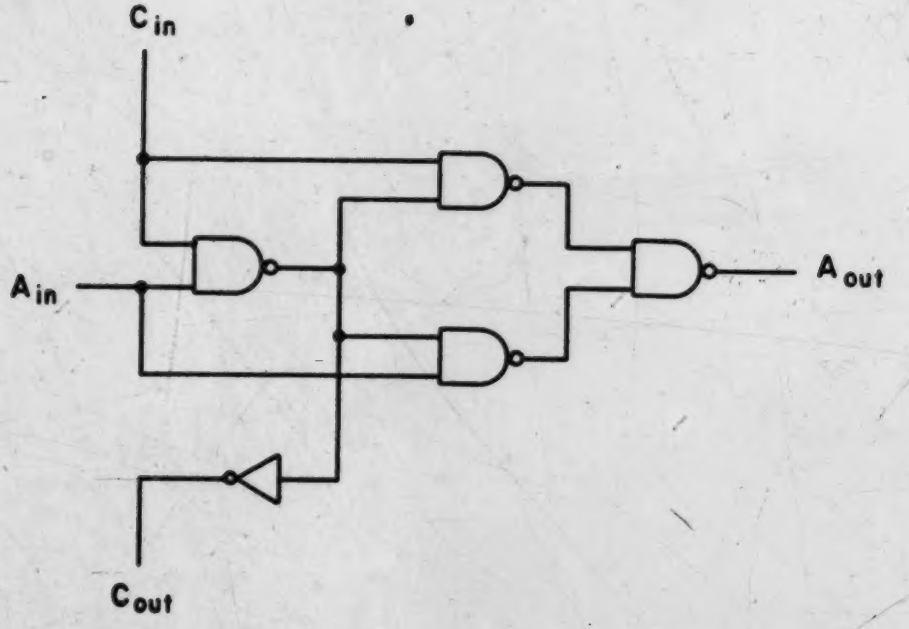


Figure 19. Gate diagram of one-bit half-adder.

the system is in the "load test" state (C1 low). This last condition will ensure that an interrupt cannot occur during a branch. This will result in the equation

$$\overline{INT} = (EINT)(IM)(\overline{RES})(\overline{C1}) ,$$

which can be implemented using one quadruple-input NAND gate.

The truth table and Karnaugh map for the lowest address bit C1 are shown in Table 7. Bit C1 will go low if a reset or an interrupt is received, or if C1 is low and $\bar{L}$ is high, signifying a failed load test, or if C1 is high. C1 will go high if C1 and $\bar{L}$ are low, signifying a branch, or if C1 is low and a subroutine call or return from subroutine is received. From the Karnaugh map, the minimised equations for $C1^{+}$ and $\overline{C1}^{+}$ (where $^{+}$ denotes the value of the signal after the next clock cycle) are

$$C1^{+} = (\overline{RES})(\overline{INT})(\overline{C1})(\bar{L}) + (\overline{RES})(\overline{C1})(CS)$$

and

$$\overline{C1}^{+} = \overline{\overline{RES}} + \overline{\overline{INT}} + C1 + (\overline{CS})(\bar{L}) .$$

These equations are derived by selecting the "1" terms and the "0" terms of the Karnaugh map, respectively. The NAND-NAND implementation of $C1^{+}$ requires a two-input, a three input, and a four-input NAND gate, and two inverters, while the $\overline{C1}^{+}$ function requires a two- and a four-input NAND gate and one inverter. Clearly, the $\overline{C1}^{+}$ function is more compact in design, as shown in the gate diagrams of Figure 20.

The equation for the high address bit (main/subroutine select) SB is derived using the truth table and Karnaugh map of Table 8. The bit will go low if $\overline{RES}$ is low, or if control is in the main program (SB low) and a subroutine is not being called (CS low), or if in the main program a subroutine is called (CS high) and C1 is low, or if in the subroutine section (SB high), a return from

$\overline{RES}$	$\overline{INT}$	C1	CS	$\overline{L}$	C1$^+$
0	1	X	X	X	0
1	0	0	0	X	0
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	X	1
1	1	1	X	X	0

a)

$\overline{RES}$ $\overline{INT}$ \ C1 CS $\bar{L}$	000	001	011	010	110	111	101	100
00	X	X	X	X	X	X	X	X
01	0	0	0	0	0	0	0	0
11	1	0	1	1	0	0	0	0
10	0	0	X	X	X	X	X	X

b)

X -- don't care

Table 7. a) Truth table and b) Karnaugh map for low address bit C1.

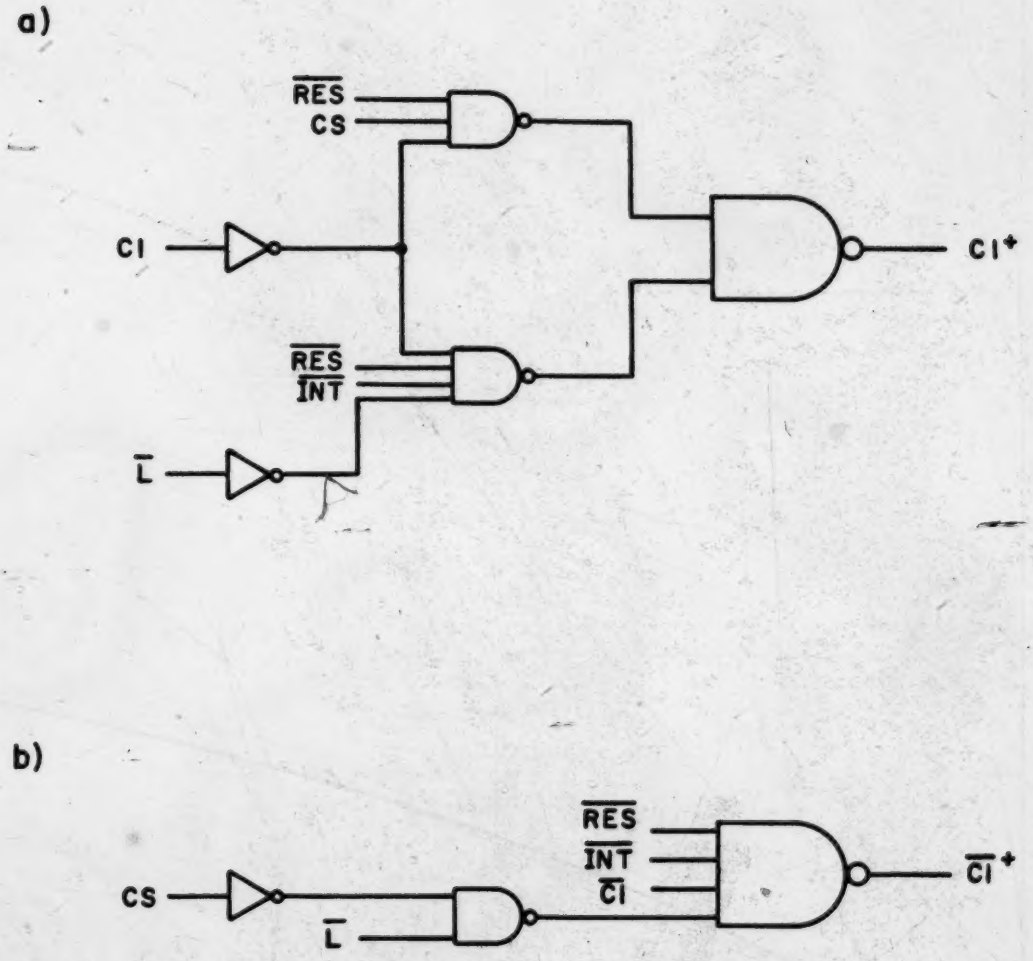


Figure 20. Gate diagrams of low address bit C1 and its inverse, $\overline{C1}$.

$\overline{RES}$	$\overline{INT}$	CS	SB	C1	SB^+
0	1	X	X	X	0
1	0	0	0	0	1
1	1	0	0	X	0
1	1	0	1	X	1
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	0

a)

$\overline{RES}$ $\overline{INT}$ \ CS SB C1	000	001	011	010	110	111	101	100
00	X	X	X	X	X	X	X	X
01	0	0	0	0	0	0	0	0
11	0	0	1	1	1	0	1	0
10	1	X	X	X	X	X	X	X

b)

X -- don't care

Table 8. a) Truth table and b) Karnaugh map for high address bit SB.

subroutine (CS high) is received and C1 is high. SB will go high if an interrupt is received, or if, in the subroutine section, return is not received, or if, in the main program, a subroutine is called and C1 is high, or if, in a subroutine, return is called and C1 is low. From the Karnaugh map, the minimised equations for SB^+ and $\overline{SB}^+$ are

$$SB^+ = \overline{\overline{INT}} + (\overline{RES})(\overline{CS})(SB) + (\overline{RES})(SB)(\overline{C1}) + (\overline{RES})(CS)(\overline{SB})(C1)$$

and

$$\overline{SB}^+ = \overline{\overline{RES}} + (\overline{CS})(\overline{SB})(C1) + (\overline{INT})(\overline{SB})(\overline{C1}) + (CS)(SB)(C1) .$$

Figure 21 shows the NAND-NAND implementation of these equations. The $\overline{SB}^+$ implementation is slightly more compact (10 versus 11 inputs).

The initial carry-in Co to the four-bit half-adder is to be high when an address increment is needed and low when the input to the half-adder is to pass unchanged. Table 9 shows the truth table and Karnaugh map for Co. Co will be high if a reset is received, or if C1 is low (indicating that a count may be needed). Co will be low if an interrupt is received, or if C1 is high (indicating the latch of a new address). The other cases are don't care conditions. The resulting equation for Co is

$$Co = \overline{\overline{RES}} + (\overline{INT})(\overline{C1}) ,$$

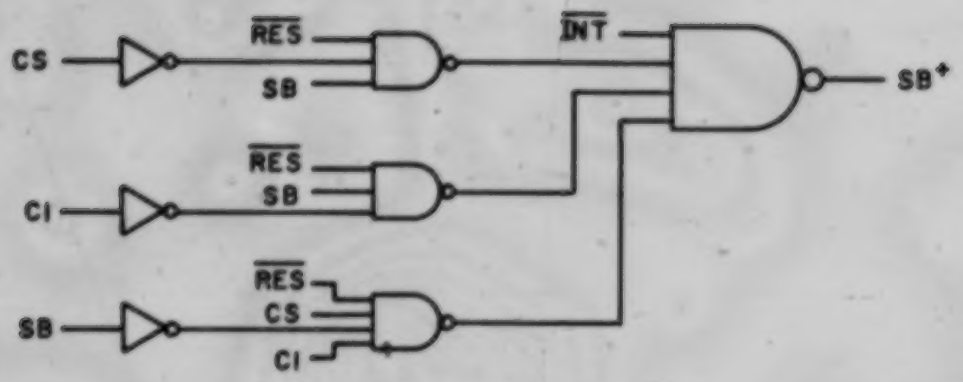
or in NAND-NAND representation:

$$Co = \overline{(\overline{RES}\ (\ \overline{(\overline{INT})(\overline{C1})}\ )\ )} .$$

This function uses two dual-input NAND gates.

The input and output registers must latch new words of information only if a branch is to take place. This means that the system can implement both Mealy and Moore finite state machines: if a Mealy FSM is desired, each transition is given a transition-specific output; if a Moore FSM is needed, all transitions to

a)



b)

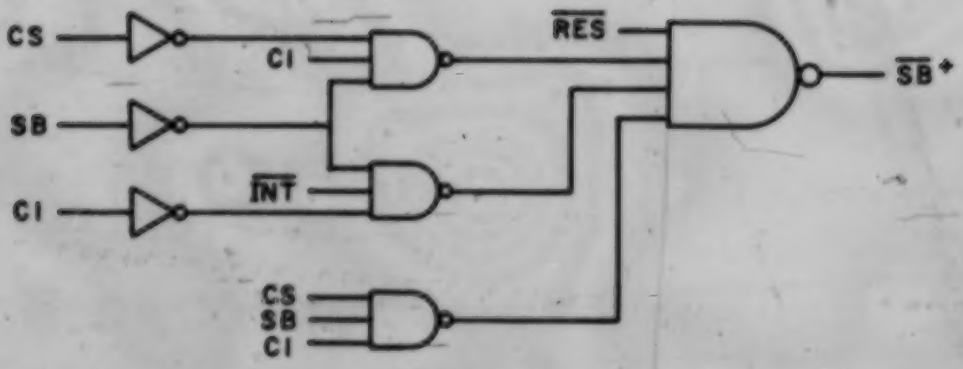


Figure 21. Gate diagrams of high address bit SB and its inverse, $\overline{SB}$.

$\overline{RES}$	$\overline{INT}$	C1	Co
0	1	X	1
1	0	0	0
1	1	0	1
1	1	1	0

a)

$\overline{RES}$ \ $\overline{INT}$ C1	00	01	11	10
0	X	X	1	1
1	0	X	0	1

b)

X -- don't care

Table 9. a) Truth table and b) Karnaugh map for initial carry-in Co.

a specific state will have the same output.

Since a branch can only occur when the low address bit is high, the I/O register latch clocks will have the following equations:

$$\text{I/O CK} = (\text{C1})(\text{CK})$$

and

$$\overline{\text{I/O CK}} = (\overline{\text{C1}})(\text{CK}) ,$$

where CK is the external clock signal. The latches will use a two-input NAND gate and an inverter.

The truth table and Karnaugh map for the return address register latch clock are shown in Table 10. The register must be latched when a return address is needed, due to an interrupt ($\overline{\text{INT}}$ low) or a subroutine call (CS high, SB low). It must not latch when a return address has already been stored for later use: during the load of the subroutine address in the main program (CS and C1 high, SB low) or during the execution of a subroutine (SB high). All other circumstances are don't care conditions since the value of the return address register is then irrelevant. The resulting minimised equation, from the Karnaugh map, is

$$\text{RADD CK} = (\overline{\text{SB}})(\overline{\text{C1}})(\text{CK}) ,$$

which will require a three-input NAND gate and inverters.

The present address register must be latched according to the truth table of Table 11.a. It must latch if a reset or an interrupt is received, or if C1 and CS are low and $\overline{\text{L}}$ is high (in order to latch the incremented address), or if C1 is high (in order to latch the new address). The register must not be latched if C1 and CS are low and $\overline{\text{L}}$ is low (indicating a successful branch will occur), or if C1 is low and CS is high (indicating a subroutine call is to take place). All other cases can never occur and are don't care terms, which will be used to minimise the equation for the latch clock. The Karnaugh map of Table 11.b shows

$\overline{INT}$	CS	SB	C1	LATCH CLOCK
0	0	0	0	1
1	0	1	X	0
1	1	0	0	1
1	1	0	1	0
1	1	1	X	0

a)

$\overline{INT}$ CS \ SB C1	00	01	11	10
00	1	X	X	X
01	X	X	X	X
11	1	0	0	0
10	X	X	0	0

b)

X -- don't care

Table 10. a) Truth table and b) Karnaugh map for return address latch clock.

$\overline{RES}$	$\overline{INT}$	C1	CS	$\bar{L}$	LATCH CLOCK
0	1	X	X	X	1
1	0	0	0	X	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	X	0
1	1	1	X	X	1

a)

$\overline{RES}$ $\overline{INT}$ \ C1 CS $\bar{L}$	000	001	011	010	110	111	101	100
00	X	X	X	X	X	X	X	X
01	1	1	1	1	1	1	1	1
11	0	1	0	0	1	1	1	1
10	1	1	X	X	X	X	X	X

b)

X -- don't care

Table 11. a) Truth table and b) Karnaugh map for present address latch clock.

that the equation for the latch clock will be just the inverse of the equation for $C1^{+}$ evaluated earlier:

$$\text{PADD CK} = (\overline{C1}^{+})(CK) .$$

This equation will require a two-input NAND and an inverter to implement.

The final signal of the control block to be evaluated is the interrupt acknowledge output, IACK. This signal should only go high (or low) when an interrupt has been accepted by the PSC. This may be achieved by tying IACK directly to the internal $\overline{INT}$ signal, which only goes low when an interrupt is to take place. The equation

$$IACK = \overline{INT}$$

will make IACK active-low and will require no additional circuitry.

2.5 Gate design

The fundamental gates from which the logic circuits are built must now be designed to minimise silicon area and propagation delays. The first step is to design the circuits at the transistor level with the least number of transistors while maintaining a reliable design. Then, the circuits will simulated using simulation software to determine the optimum size of each transistor and to verify the function of each circuit.

The gates to be designed are the inverter, the two-input NOR, the two-, three-, four-, and five-input NAND, the EXOR, and the transmission gate used by the data register.

In CMOS technology, the simplest and most reliable method of logic gate design is the use of p-transistors to raise the output signal and n-transistors to lower the signal. This means that, for the inverter circuit shown in Figure 22.a, a low input will turn the p-transistor on and produce a high output, while a high input will turn the n-transistor on and produce a low output. Similarly, the two-input CMOS NAND circuit of Figure 22.b requires either input A or B low to pull the output high, and both inputs A and B high to pull the output low. Multi-input NAND circuits will operate on the same principle. Figure 22.c shows the two-input CMOS NOR circuit, whose output will be pulled high if both inputs are low, and will be pulled low if either input is high.

The EXOR circuit is more difficult to design. There are several acceptable designs: the four NAND gate half-adder circuit of Figure 16 (which would require sixteen transistors); the two-level AND-OR implementation

$$A \oplus B = (\overline{A})(B) + (A)(\overline{B})$$

(twelve transistors); and the six-transistor design of Figure 23.a. The operation of the latter circuit is very simple [24]. If input A is low, the transmission

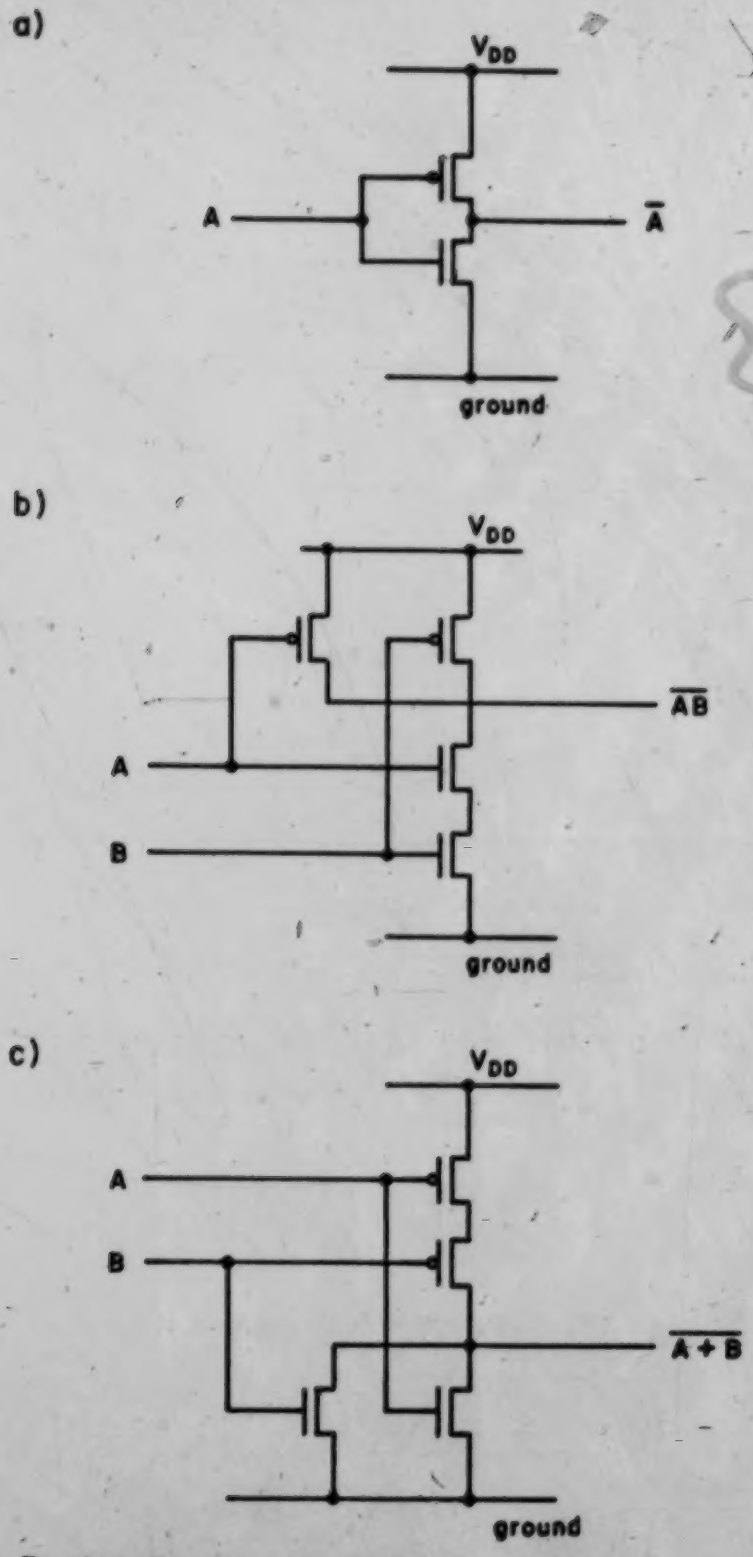


Figure 22: Transistor diagrams of inverter, NAND, and NOR gates.

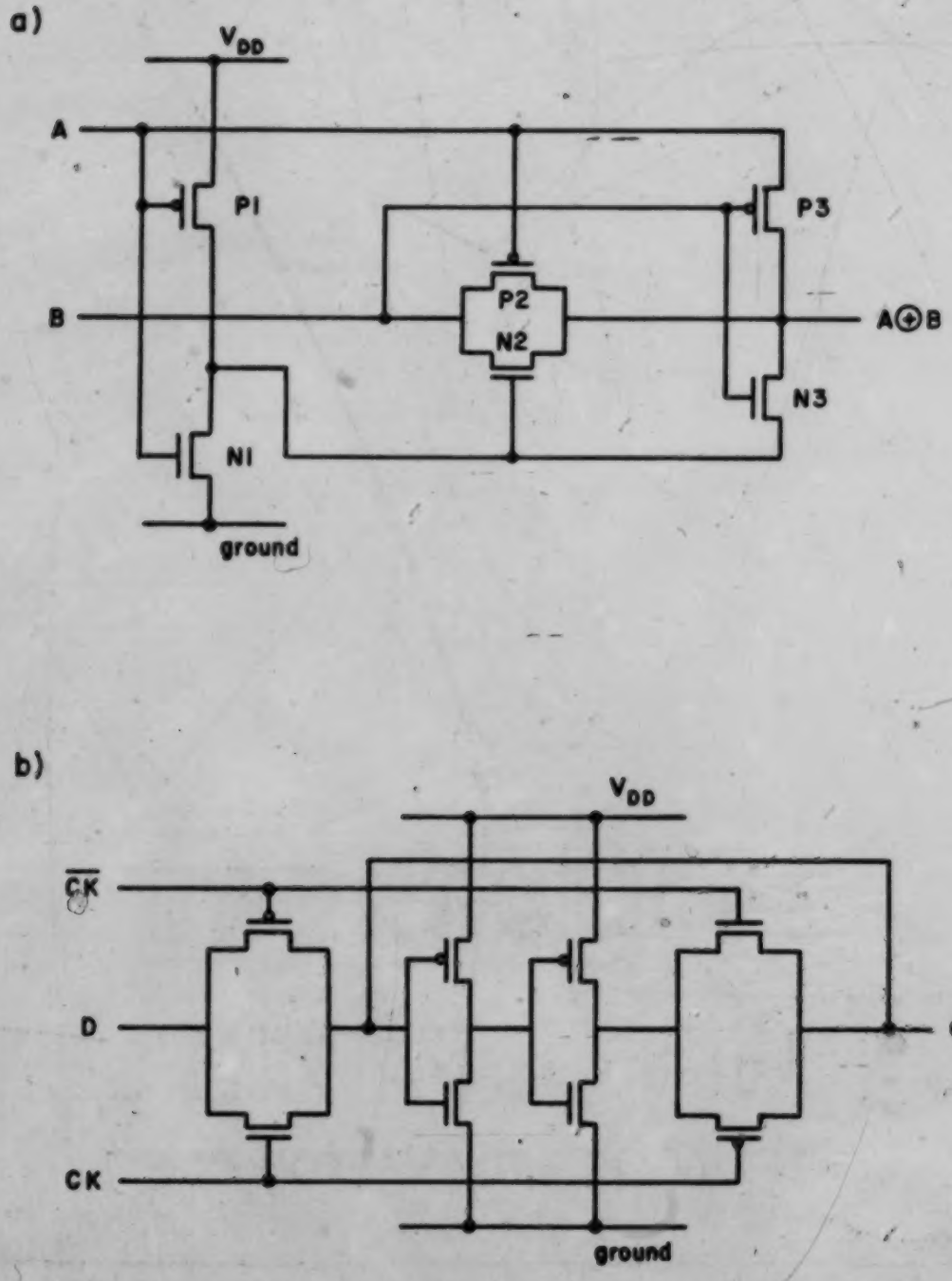


Figure 23. Transistor diagrams of EXOR gate and data register.

gate (formed by two complementary transistors P2 and N2) is turned on, so that the B input is passed to the output. If A is high and B is low, A is passed through P3; if A and B are high, $\overline{A}$ is passed through N3. This design is much smaller in area than the conventional designs, and, because it has only two logic levels, it will be at least as fast as the other designs.

Figure 23.b shows the CMOS transistor circuit implementation of the data register. It uses two CMOS inverters, as described above, and two CMOS transmission gates. The transmission gates are just two complementary transistors in parallel, with drains connected to the input and sources connected to the output. Both transistors are enabled simultaneously by complementary signals, CK and $\overline{CK}$, because the p-transistor passes high signals better, while the n-transistor passes low signals better. The register circuit requires a total of eight transistors This is a good improvement over the flip-flop register, which requires sixteen transistors (NAND implementation).

The transistor design of the IC is now complete. The present design will require approximately 120 transistors for the logic block, 350 for the address block, and 150 for the control block, for a total of over 600 transistors for the whole circuit. This defines the IC as a small LSI circuit.

2.6 Circuit simulations

The circuits which have been designed can now be simulated at the transistor level on a computer. There are two principal purposes in performing a simulation. The first is to verify that a particular circuit functions as desired. The second is to improve the circuit by determining the optimum transistor dimensions.

The circuit simulation program to be used is called SPICE [37]. A typical SPICE run requires an input file consisting of transistor parameters (such as electron mobility), a nodal description of the circuit to be analysed, and the specification of voltage levels on the input signals as a function of time. The transistor parameters depend on the silicon technology used to fabricate the IC; in this case it is the NT CMOS 1B 5 micron process. The SPICE program will simulate the circuit using these parameters and determine voltage levels at specific nodes as a function of time.

The function of a given circuit can thus be verified by applying all possible combinations of inputs and monitoring the outputs. This will be particularly useful in determining the operation of the transmission gate, data register, and EXOR gate, since it is not readily obvious whether these these circuits will function correctly.

Propagation delay is defined as the time taken for an output signal to reach 50% of its final value from the time when the input signal reached its 50% level. Rise and fall times measure the time taken for an output signal to rise from 10% to 90% of its final value or fall from 90% to 10% of its final value. By designing a circuit to minimise and equalise rise and fall times, the worst-case propagation delay can be minimised as well.

Because the n-transistor and p-transistor have different mobilities (a ratio of about 3 to 1), the rise and fall times of a circuit cannot be equalised

by choosing minimum dimensions for all transistors. Instead, it is standard to choose minimum dimensions for the faster half of the circuit and then choose the dimensions of the slower half in order to equalise the rise and fall times. Since the rate of charge transfer varies with the width of the transistor, transistors of the slower half must be widened to achieve the optimum dimensions. The circuits to be modelled will be the basic gates designed in the previous section: the inverter, NAND, NOR, transmission gate, data register, and EXOR gate. The half-adder circuit will also be modelled to aid in determining the overall propagation delays within the IC.

Table 12 shows the complete SPICE results for the worst case rise and fall times and propagation delays for the modelled circuits for different p- to n-transistor width ratios (all transistors have minimum length of five microns since the speed of a transistor varies inversely with its length).

The inverter operates as expected, showing the best results for a p to n ratio of sixteen to five (equal to the mobility ratio of the transistors). This inverter has rise and fall times of about two nanoseconds (ns) and propagation delays of 1.4 ns.

The two-input NAND also operates correctly. The worst case rise time occurs when only one p-transistor is on, while the only possible fall time case occurs when both n-transistors are on, doubling the effective transistor length. These two times can thus be equalised by choosing a p to n width ratio of eight to five; this produces rise and fall times of about five ns and propagation delays of two ns. Similarly, the three-input NAND has an optimum p to n width ratio of about one to one, with rise and fall times of 6.5 ns and propagation delays of three ns. The four-input NAND has an optimum p to n ratio of five to six, with rise and fall times of seven ns and propagation delays of about 3.5 ns.

GATE	P:N RATIO	RISE TIME	FALL TIME	RISE DELAY	FALL DELAY
INVERTER	16: 5	1.9	2.0	1.4	1.3
NAND: TWO-INPUT	8 : 5	4.8	4.0	2.0	1.9
THREE-INPUT	1 : 1	6.5	6.5	3.0	3.0
FOUR-INPUT	1 : 1	7.0	9.0	3.5	3.5
FOUR-INPUT	5 : 6	7.0	7.0	3.5	3.0
NOR: TWO-INPUT	16: 5	4.2	2.2	2.4	1.5
TRANSMISSION GATE	16: 5	2.5	1.0	X	X
	1 : 1	2.0	1.0	X	X
EXOR GATE	16: 5	1.2	2.0	X	X
	1 : 1	2.4	3.6	X	X
REGISTER	16: 5	4.0	4.0	X	X
HALF-ADDER:					
FIRST BIT CARRY	X	X	X	5.5	X
FIRST BIT SUM	X	X	X	11.5	X
SECOND BIT CARRY	X	X	X	9.0	X
SECOND BIT SUM	X	X	X	15.5	X

All times are in nanoseconds.

X -- not calculated.

All dimensions are minimised (5 micron gate width), save the EXOR gate with 1:1 ratio. This circuit has p- and n-transistor dimensions of 9:5 (width to length).

Table 12. Worst case rise and fall times and propagation delays for various circuits and transistor dimensions using SPICE.

The worst case fall time for the two-input NOR will occur when only one n-transistor is on, while the worst case rise time will be when both p-transistors are on. Therefore, the optimum p to n width ratio would probably be about six to one. However, it is not practical to design a p-transistor of a width of 32 microns; instead, a p to n ratio of sixteen to five is chosen. This gives a rise time of 4.2 ns, a fall time of two ns, and propagation delays of 1.5 and 2.4 ns, respectively. Since there are only five NOR gates in the whole IC, this is quite acceptable.

The operation of the transmission gate is shown in Appendix 1.b. The gate operates as desired; however, it appears to perform slightly better with minimum-dimension transistors than with a three to one p to n ratio. The rise time for a one to one ratio is two ns, while for a three to one ratio the rise time is 2.5 ns. The fall time in both cases is one nanosecond.

The SPICE results for the EXOR gate of Figure 23.a are shown in Appendix 1.c. The circuit operates as expected. However, the rise and fall times differ for each input combination because, for each combination, the circuit operates very differently. For example, when both inputs A and B are low, the output is pulled low by B through the transmission gate and by A through P3. When both A and B are high, the same output is produced by the inverter through N3. The worst case occurs when A is high and B is low: the output is pulled high only by A through P3. Thus, P3 is widened to reduce the worst case delay, while N3 is designed with minimum dimensions.

Appendix 1.d shows the operation of the data register, using inverters with a three to one p to n ratio and transmission gates with a three to one ratio. The operation is as desired, with the output being passed on the clock high, and held on the clock low. The rise and fall times are found to be about four ns.

Since a single clock signal is being used, there may be a slight overlapping of the clock signal CK with its inverse signal $\overline{CK}$, due to the propagation delay of the inverter. It is important to ensure that this does not create latching problems [19]. Using SPICE, the input to the register is changed shortly after the clock has latched the last input. When the input changes one nanosecond after the latch, a "spike" appears on the output of the register; however, the operation is still correct. When the input changes half a nanosecond after the latch, the "spike" becomes more pronounced, but the register continues to operate as desired.

An examination of the circuit design shows that no register output (such as C1, SB, or the present address) is fed back to the register input through fewer than two gate levels. This will delay the change of input due to a change in output by at least one ns, so that the registers should function properly without the need for a two-phase, non-overlapping clock signal.

A complete sample SPICE run of the data register model is shown in Appendix 1.a.

The half-adder circuit of Figure 19 was modelled on SPICE for a two-bit circuit using optimum inverter and NAND dimensions. The worst case occurs when all input signals are high: this forces the first-bit carry to propagate through the maximum two gate levels, the first-bit sum through three gate levels, and the second bit-carry and sum through the maximum four gate levels. The corresponding propagation delays are 5.5 ns, 11.5 ns, 9.0 ns, and 15.5 ns, respectively. It is thus estimated that, at worst, the time required for an n-bit half-adder sum to propagate would be

$$T_{adder} = 5n + 10 ,$$

in nanoseconds.

From the completed SPICE results, an estimate of the performance of the

IC can be made. There are five major timing concerns to be examined: the test cycle, the load/increment cycle, the interrupt, the subroutine call, and the return from subroutine.

The test cycle compares the masked input to the input condition to produce $\bar{L}$ and then calculates the new value of C1. By examining the gates through which the signal must propagate in an N-chip circuit, the propagation delay during the test cycle is found to be

$$T_{test} = T_{nor} + T_{exor} + T_{nand4} + N(T_{nor} + T_{inv}) + T_{C1}$$

where the NOR delay is about three ns, the EXOR delay is four ns, the four-input NAND delay is four ns, the inverter delay is two ns, and the delay to evaluate C1 is

$$T_{C1} = T_{inv} + T_{nand2} + T_{nand4} + 2(T_{reg}),$$

where the two-input NAND delay is about two ns and the register delay is four ns. Thus, the C1 delay is about 16 ns, and the test cycle delay is about (5N + 27) ns for an N-chip circuit. This delay is 47 ns for a four-chip, 16-bit system.

The load/increment cycle either loads a new address or increments the present address, depending on the state of the counter C1. The signal must propagate through the MUX5, half-adder, and present address register. The propagation delay is given by

$$T_{load} = T_{mux5} + T_{co} + T_{adder} + 2(T_{reg}),$$

where the initial carry-in delay is

$$T_{co} = T_{C1} + 2(T_{nand2}) = 19 \text{ ns},$$

and the adder delay is about (20N + 10) ns, and

$$T_{mux5} = 3(T_{nand2}) + T_{nand5},$$

where the five-input NAND delay is about five ns. Thus the MUX5 delay is about 11 ns, and the load/increment cycle delay is about (20N + 33) ns, for an N-chip circuit.

The interrupt cycle involves four parallel processes: 1111 is passed from the MUX5 to the present address register, the present address is latched on the return address register, C1 is kept low, and SB is set high. The delay to load 1111 will be at least as short as the delay of the load/increment cycle and the C1 delay is 16 ns. The delay to latch the return address is given by

$$T_{radd} = T_{mux2} + 2(T_{reg}),$$

where

$$T_{mux2} = T_{inv} + 2(T_{nand2}),$$

so the MUX2 delay is about six ns, and the return address delay is 14 ns. The high address bit SB delay is

$$T_{sb} = T_{inv} + T_{nand3} + T_{nand4} + 2(T_{reg}),$$

or about 17 ns. These delays occur in parallel, so the timing is dominated by the longest delay: the latch of 1111. Since this delay is shorter than the load/increment cycle delay, the interrupt cycle timing will be of no further concern.

The call subroutine cycle consists of two separate cycles. On the first clock pulse, the return address is latched and C1 is set. On the second pulse, the new address is latched, SB is set, and C1 is cleared. The first cycle is dominated by the C1 delay, which is only 16 ns. The second cycle is dominated by the new address latch, which is at least as fast as the load/increment cycle.

The return from subroutine cycle also consists of two clock cycles. On the first cycle, C1 is set. On the second cycle, the return address is latched,

and C1 and SB are cleared. Again, these cycles are at least as fast as the load/increment cycle.

Therefore, the maximum length of time required for the N-chip system to operate during one clock cycle is

$$T_{system} = T_{load} = 20N + 33 ,$$

in nanoseconds. This figure is doubled to allow for possible errors and to ensure an accurate worst case delay. This means that an N-chip PSC system can be run at a maximum clock frequency,

$$f = \frac{100}{4N + 7} \text{ MegaHertz (MHz).}$$

For one chip (four bits), this frequency is about 9 MHz; for two chips (eight bits), it is about 7 MHz; for four chips (16 bits), it is about 4.5 MHz. These maximum frequencies are quite acceptable for most industrial applications. If greater speeds are desired, the focus of the design should be to improve the speed of the half-adder carry propagation, since this is the dominating timing concern. The simplest solution is to increase the transistor widths of the two-input NAND and inverter gates that produce the carry-out for each half-adder stage.

2.7 Circuit layout

The designed circuit must now be described graphically in the form of a layout. This layout will consist of sets of rectangles of eleven different layers needed for the NT CMOS 1B fabrication process: p-well, device well, p guard, n guard, polysilicon, capacitor polysilicon, p^+ mask, n^+ mask, contact cut, metal, and glass. The rectangles will define the transistors and interconnections of the PSC circuit. The layout will then be translated into a set of instructions in an intermediate language which may be used to fabricate the integrated circuit.

The graphics editor used is called KIC [3]. It allows the user to describe rectangles of different layers by selecting a layer and pointing to two opposite corners of the rectangle, using a "mouse" and a high-resolution graphics display terminal. KIC also permits the user to copy, move, reflect, and rotate the rectangles. Supporting software allows the user to perform rule checks, obtain plots, and convert the KIC layout into descriptive language programs.

In laying out the circuit, it is desired to make the design as compact as possible. There are, however, cetain limits on the compactness of the design. These limits are called design rules. They set the minimum separation of two rectangles of the same layer, the minimum overlap of one layer over another, and the minimum size of a rectangle of a given layer. These rules are a function of the technological process being used. For the present design, the five-micron process wil be used.

The layout begins with the design of a CMOS inverter. It will consist of an n- and a p- transistor connected in series, with the same input on the gate of each transistor. The traditional way to design this circuit is to lay out the transistors vertically, as shown in the "sticks" diagram of Figure 24.a (a sticks diagram shows only the metal, poly, device well and p-well rectangles of a layout).

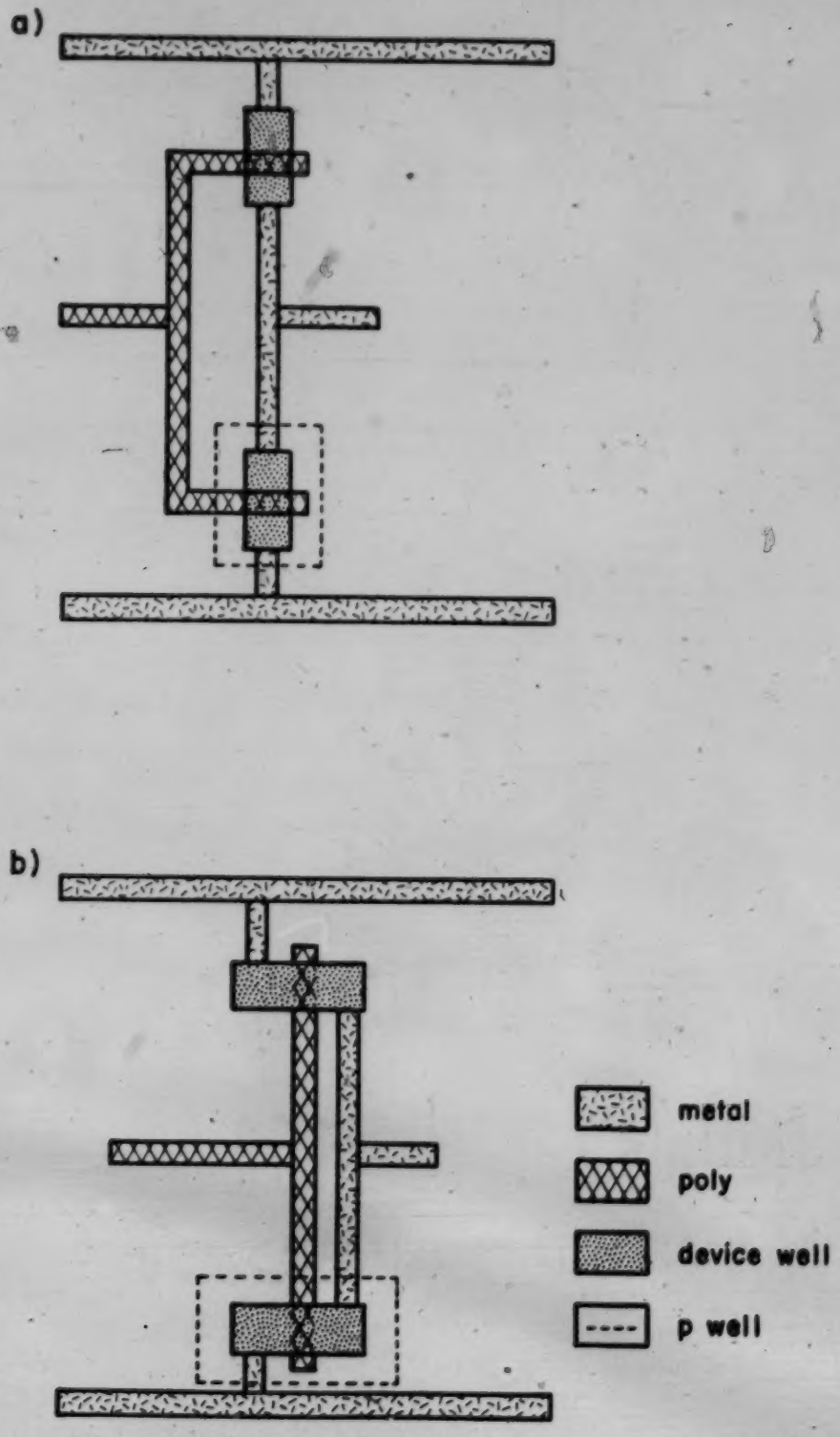


Figure 24. Stick diagrams of inverter.

However, this circuit will be tall and thin, especially when it is stretched to allow room for other signals to pass horizontally through the middle. The design of Figure 24.b has the transistors layed out horizontally, leaving more room for horizontal signals and leading to a compact and square design.

The other gates are designed in a similar manner. The two-input NAND, for example, will have two transistors lying horizontally at the top and the bottom of the layout. The p-transistors will be connected in parallel, while the n-transistors will be connected in series. The gates are layed out individually in a work-space area to form a "library" of circuits. These circuits can then be selected and copied into larger circuits as required.

As each gate is designed, it is checked for layer rule violations (LYRV). KIC contains a layer rule checker program (LYRA) which determines whether any rules have been violated. It is very fast and simple to use, and saves hours of inspection time for LSI circuits. If any violations are found, they can be corrected immediately, before they are duplicated in more complex gates.

Now a complete level of a functional block is layed out and connected to horizontal power and ground lines. The logic and address blocks are four bits wide; therefore, for these blocks, the level must be copied and mirrored three times, resulting in four identical levels of gates. Mirroring causes the power and ground lines to change places, allowing the IC to take advantage of the combed power and ground grid. Because two levels share each power or ground line, these lines are given a ten micron width (twice the minimum metal width).

When a complete functional block has been layed out, it is again checked for LYRVs. As the area of the layout being checked increases, the time taken to run LYRA increases enormously.

After all three functional blocks have been layed out and placed in

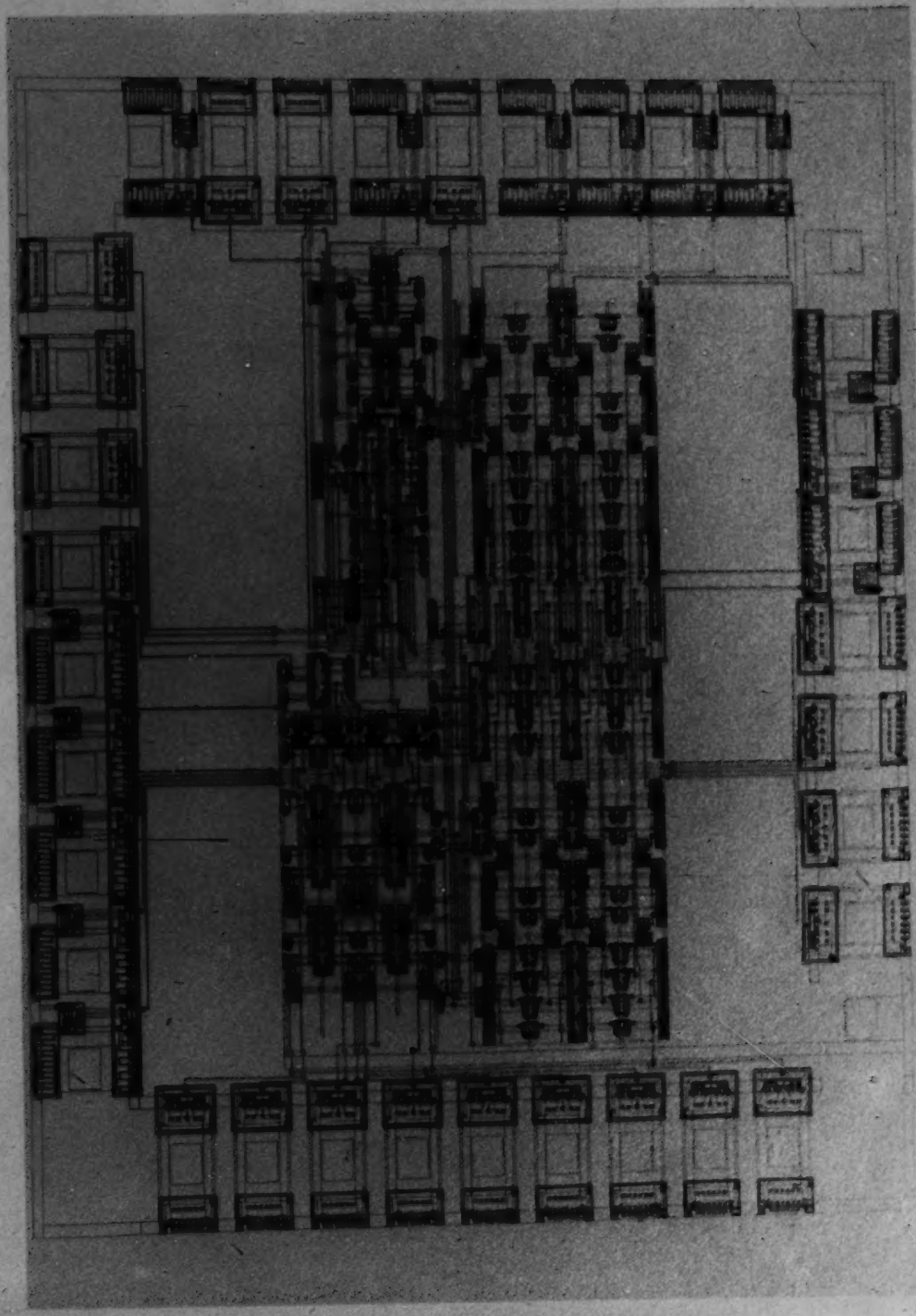
position with respect to one another, interconnections between blocks are made. This involves the feeding of signals such as latch clocks to the proper registers, and the connection of data buses such as PROM lines to the proper logic gates. Most signals are routed between the functional blocks, or through them at specific areas, as shown in the floor plan of Figure 16. The separation between signal lines is kept to the minimum (5 microns for polysilicon, 6 microns for metal), in order to reduce the area of the layout.

Finally, the input, output, power, and ground pads are layed out along the perimeter of the IC. These pads will be used to connect the circuit inputs and outputs to the package pins. As well, the input pads are designed to protect MOSFET transistors against static electricity discharges. Output pads are designed as drivers, consisting of a series of successively wider inverters. These pads allow the circuit to drive greater loads than would otherwise be possible The power and ground pads are simple metal pads. They are connected to the internal power and ground grid and the input/output pads by means of power and ground lines encircling the IC. In order to reduce the power density on these principal lines, they are made very wide (30 microns).

After all pads have been connected to their respective signal lines and to the encircling power and ground lines, a final rule check is performed and all remaining LYRVs are corrected. The completed programmable sequential controller IC layout is shown in Plate 1. The layout has dimensions of 3210 microns by 3210 microns (about ten square millimeters). Since this is the minimum size possible for an IC with 36 pads on the perimeter, the design has been made as compact as was necessary.

The KIC layout is converted to a file in a layout language, which describes all the rectangles and layers found in the KIC layout. This file is sent to the

Plate 1. Integrated circuit layout.



CMC, where the design is placed on a multi-project board and fabricated in four to six months.

III A LOW-LEVEL LANGUAGE ASSEMBLER

3.0 Introduction

The PSC can be microprogrammed by describing five words of information for each branch in a finite state machine: the input mask, input condition, next state, output, and control word. The microprogramming can be performed at several levels of program specification [18,27].

The lowest of these is the "machine language" level. A machine language program specifies exactly the bytes to be used by the controller to implement the desired FSM. This type of program can be loaded directly into the PROM memory without any intermediary translation. However, for large FSMs, programming in machine language can be tedious and time consuming.

A "low-level" language uses "pseudo-English" instructions to describe each branch in the FSM. A program in this language may be written directly from the state diagram, but will require translation to the machine language level. The program will be shorter than the machine language program while the language syntax will be relatively simple.

At the highest level, the language will use complex instructions (such as "WHILE" loops) and procedures to describe the FSM from a systematic viewpoint. There will be no direct relationship between instructions and branches in this "high-level" language program. This means that the language will have more complex syntax and will require a large amount of translation in order to produce the equivalent machine language code.

In most instances in control circuit design, the designer will proceed from the problem specification to the FSM diagram. At this stage, the optimum design

language will be the one which allows the direct translation of the FSM into program form. This will be the low-level language which uses one or two simple instructions per branch. PSC programs in this language will be easier to read and write than programs in other languages. The designer will be required to understand neither the internal operation of the PSC necessary for machine language programming nor the complex syntax of a high-level language.

A translation program that takes as input a program in a higher-level language and produces as output the equivalent program in a lower-level language is called a compiler [21]. A compiler will be written to take low-level PSC language programs and produce programs in PSC machine language. This type of compiler, which converts a low-level language to machine code, is known as an assembler.

3.1 Assembler specifications

The assembler program should meet certain specifications. It should take a

The assembler program should meet certain specifications. It should take a file containing PSC low-level language instructions and decode them into machine language instructions which can then be loaded into the three external PROMs. It should also check the input file for syntactical and logical programming errors and supply messages to assist in the correction of these errors.

All branches of an FSM can be described using the following five low-level instructions [22]:

1. IF defines a conditional branch to another node.
2. GOTO defines an unconditional branch to another node (actually a special type of IF instruction).
3. JUMP defines a subroutine call.
4. RTS defines a return from subroutine.
5. OUT defines the state of the outputs after a successful branch.

No input instruction is needed, as the inputs are latched synchronously after every successful branch.

In addition, the following identifiers are required in order to completely describe the FSM:

6. NODE defines a unique node in the state graph. All instructions following a NODE instruction refer to branches from that node.
7. SUBROUTINES is a label separating main routine nodes from subroutine nodes.
8. MASK is a label indicating a node which is to be masked from interrupts.
9. VECTOR INTERRUPTS defines the initial node of the interrupt service routine.

Other instructions affecting the control PROM which may be added include SET

TIMER to program an external timer and SELECT INPUT/OUTPUT PORT to control I/O flow. These instructions are not related to the internal of the PSC chip; they only affect external circuitry to be added by the user. Therefore, such instructions may be added to the language by the user/programmer.

Each instruction in the input file (save the NODE and SUBROUTINES instructions) should be translated into the equivalent machine language information and stored in three distinct output files (which will later be loaded into the PROMs). For example, the MASK instruction will result in a "1" being placed in bit 7 (the high bit) of the control bytes which are assigned to the node preceding the MASK instruction. Table 13 shows the effect of each instruction on the three PROMs at the bytes assigned to a given node.

The NODE instruction must be used to assign a PROM address to each node. Each node must be given a unique address, for it is at this address that the branching information will be stored.

The SUBROUTINES instruction should be used to distinguish between main program nodes (placed before this instruction) and subroutine nodes (placed after this instruction). This will be done by assigning addresses in the bottom half of PROM memory to main program nodes and in the top half to subroutine nodes (since subroutines are run with the high address bit set high).

COMMAND	PROM 1 BYTE1	PROM 1 BYTE2	PROM 2 BYTE1	PROM 2 BYTE2	CONTROL PROM BIT7	CONTROL PROM BIT6
IF	INPUT MASK	OUT-PUT	INPUT COND.	NEXT ADDR.	X	0
GOTO	$FF (HEX)	OUT-PUT	$FF	NEXT ADDR.	X	0
JUMP	X	OUT-PUT	X	SBRT. ADDR.	1	1
RTS	X	OUT-PUT	X	X	1	1
MASK	X	X	X	X	1	X
* VECTOR INTERRUPTS	$FF	$00	$FF	SBRT. ADDR.	1	0

X -- Command has no effect on PROM.

* -- Command has effect at addresses $1FE and $1FF.

Table 13. Effect of PSC language command on three external PROMs.

3.2 Assembler design

The assembler program is designed using a "top-down" approach. The problem is decomposed into subproblems which solve major tasks; these subproblems are further broken down, until the subproblems are small and simple enough to be converted directly into programming language code.

The assembler program reads an input file containing the low-level program, transforms or decodes this program into machine code instructions, and produces the output files and error messages. The transformation, as shown in the structure diagrams of Figures 25 and 26, consists of two tasks: first, each node in the program is assigned an address in PROM, and then the instructions of each node are translated into machine code and placed at the appropriate memory location.

The first task, "GET ADDRESS", is performed by iteratively reading a line, recording its line number (to be used in error messages), and then testing the line. If the first column of the line is an "N", then the instruction should be "NODE".

To assign the node an address requires three functions: the instruction syntax is checked; the node name (the three characters immediately following "NODE") is compared to previous names for uniqueness, and the name is stored and assigned the next available sequential address. Errors are generated if the syntax or uniqueness is violated.

If the first column of the line is the first letter of a valid instruction, the address is affected according to the nature of the instruction. If an "IF", "GOTO", "JUMP", or "RTS" instruction is encountered, the address is incremented by one to permit the instruction to be stored in program memory during the translation task of the assembler. If the instruction is "SUBROUTINES", the address is set to zero as at the start of the program, to mark the start of the

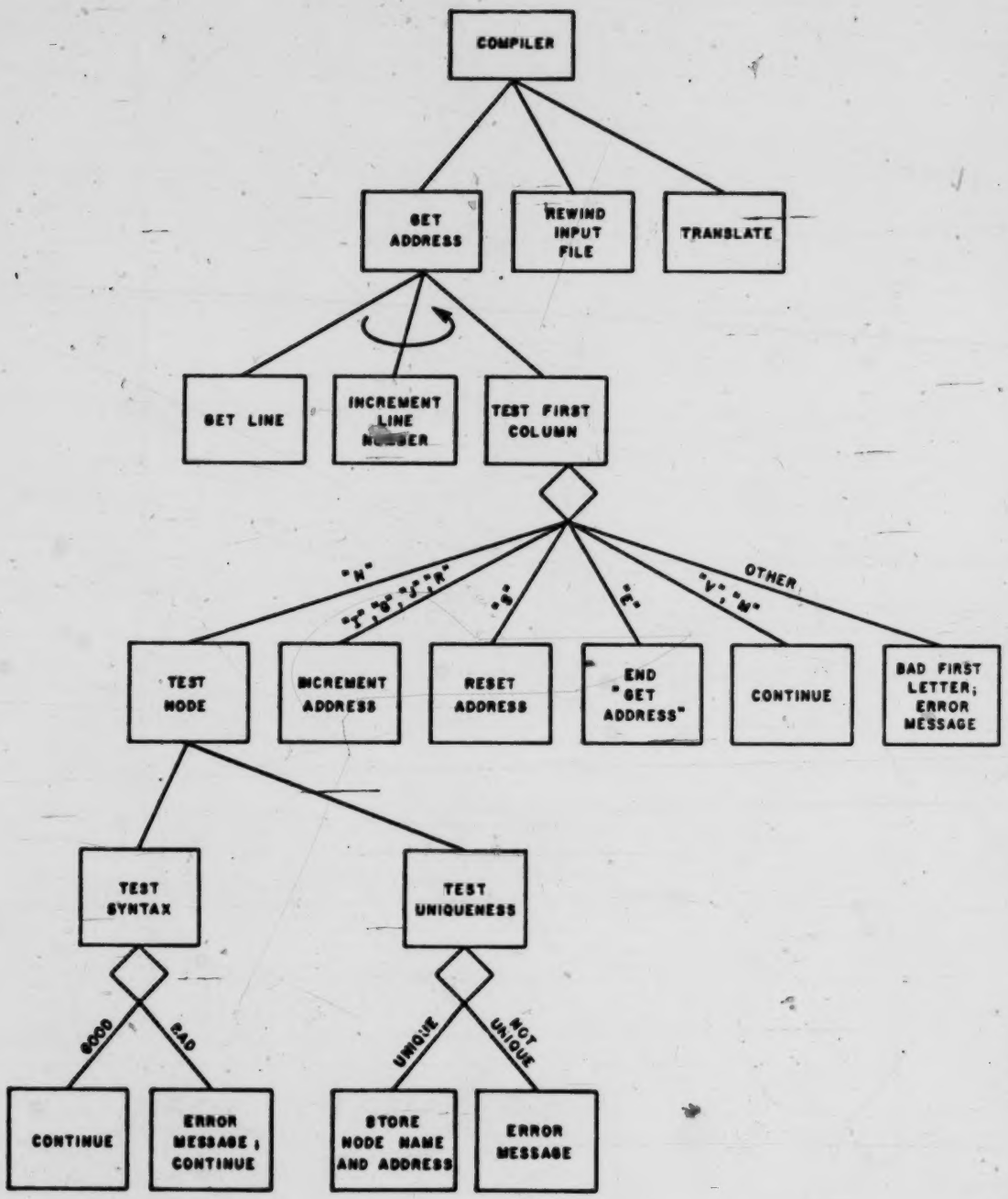


Figure 25. GETADDRESS branch of assembler program structure tree.

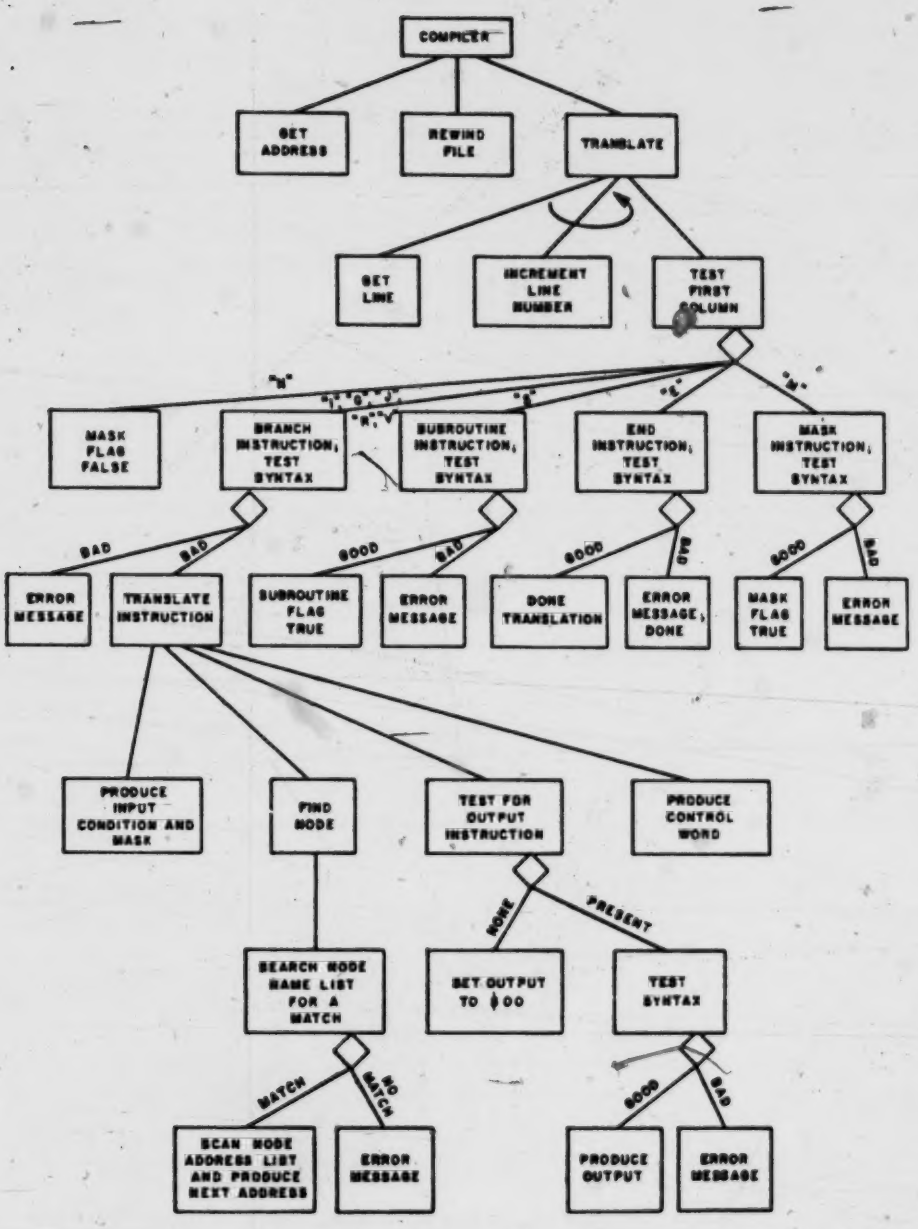


Figure 26. TRANSLATE branch of assembler program structure tree.

second half of memory. If the instruction is "END", the "GET ADDRESS" task has been completed. If the instruction is "MASK" or "VECTOR INTERRUPTS", the address is not affected. If the line begins with an illegal letter, an error message is produced. As for all error occurrences, the program continues its execution to completely analyse all input instructions.

The second assembler task, "TRANSLATE", is performed by iteratively reading a line of the input program, incrementing the line number, and translating the input instruction as shown in Table 13. The translation begins with a test of the first column of the line.

If the first column is "N", the command is NODE, so no translation is necessary. However, the interrupt mask flag is set to "false", to indicate that, unless a MASK instruction is detected before the next NODE, interrupts should not be masked in this state.

If the first column is one of "I", "G", "J", "R", or "V", a branch instruction is indicated. The syntax of the instruction is checked. If it is valid, the input mask and condition are produced. For GOTO, JUMP, RTS, and VECTOR INTERRUPTS, this is simply a matter of setting them to $FF. For the IF instruction, however, the input word consisting of 0's, 1's, and "don't cares" must be read, tested, and translated. Now the destination node is decoded to produce the next address word. The name of the destination node is read, tested, and matched to a name in the the node name list produced by the "GET ADDRESS" task. If a match occurs, the next address is taken from the node address list. Otherwise, an error message is produced. The branch instruction is then checked for the optional output instruction, "OUT". If one is found, the output word is produced as specified. Otherwise, a zero ($00) output word is produced. Finally, the control word is produced by checking the interrupt mask and subroutine call

flags and optional I/O control flags.

If the first column is an "S", the instruction will be "SUBROUTINES", so no translation will be necessary. The instruction is tested for syntax, and, if it is valid, the subroutine flag is set to "true" to indicate that all instructions following must be subroutine instructions. This will provide a means of determining that all JUMP instructions appear in the main routine and all RTS instructions appear in the subroutine section.

If the first column is an "E", the line is tested for a valid "END" instruction. If it is valid, the translation task has been completed.

If the first column is an "M", the line is tested for a "MASK" instruction. If it is valid, the mask flag is set to "true." In this way, interrupts will be masked during the execution of all instructions that follow the "MASK" command in the present node.

If the first column is any other character, an error message is produced.

The algorithm is implemented in AlgolW, a powerful high-level language for textual manipulations. The assembler program listing, with documentation detailing program compilation and execution, is shown in Appendix 2. The assembler is run with several input programs which test all aspects of the PSC language, and all minor errors in the assembler are corrected. A sample PSC language program and the three resulting assembler language files are shown in Appendix 3 (these files are then translated into machine language using an existing cross-assembler program).

IV INTEGRATED CIRCUIT TEST

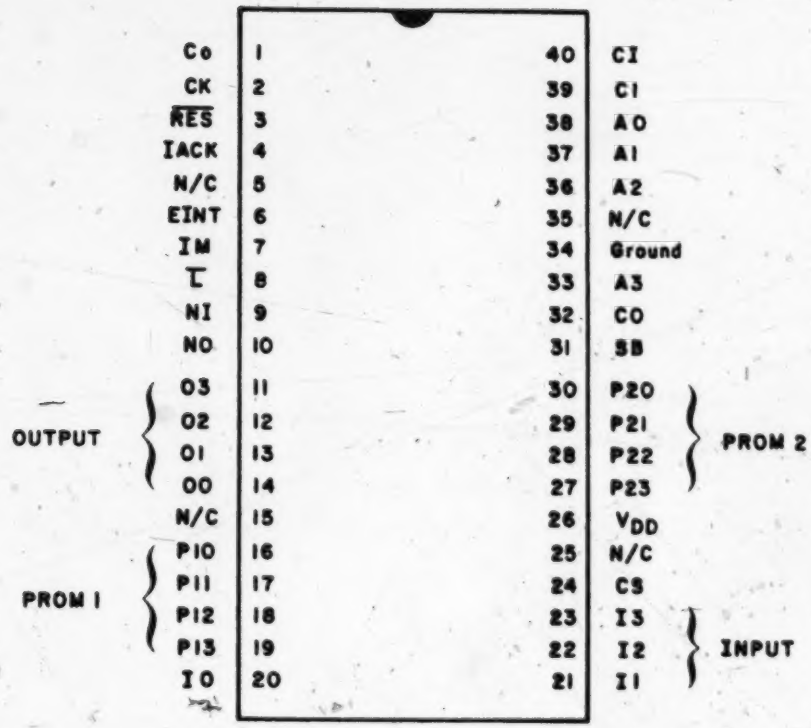
4.0 Introduction

Five packaged programmable sequential controller ICs were received from the CMC. The pin-out diagram of the chip is shown in Figure 27. The forty-pin package includes 34 I/O pins, power and ground pins, and four extra pins with no connection to the internal circuit.

The final step in the integrated circuit design process is the testing of the fabricated circuit. If any design errors are discovered or any other design modifications desired, the process returns to an earlier design step and the cycle is repeated.

The testing of an LSI circuit such as the PSC chip is difficult for two reasons. The first is that the IC contains hundreds of transistors and thousands of interconnecting lines, yet only 34 I/O pins are available to determine where any faults in the circuit design or fabrication may lie. The second is that, in order to completely test such a complex circuit, extremely long and complex test patterns must be generated.

These difficulties will be solved or lessened by the use of microprocessor-based algorithmic test generation and both explicit and implicit testing techniques [1,25,26]. Explicit testing involves the removal of the test process from the normal operation of the circuit; implicit testing is performed during the normal operation.



N/C - no connection

Figure 27. Pin-out diagram of PSC 1C.

4.1 Explicit testing

Before the chip can be operated under normal conditions, it should be tested independently of its normal operations, to determine what gates, sections, or blocks contain faults in design or fabrication. In this way, external modifications made to correct these faults temporarily may allow the normal operation of the chip to proceed.

The logic block will be tested first. This block consists of thirteen inputs (input nybble, mask nybble, condition nybble, and an earlier chip's load signal NI), and just one output, the load signal NO. In other words, the function NO must be tested for about 8000 input combinations. This test will require a microprocessor-based algorithm.

The algorithm is quite simple. The microprocessor outputs a thirteen-bit combination to the PSC through a parallel port (PIA). It then calculates the desired value of the function NO and compares it to the PSC result, read in through the PIA. If the two bits disagree, the input combination is stored in memory. This test is performed for all 8K possible input combinations. The program implementing this algorithm (written in assembler language for an MC6809 microcomputer) is shown in Appendix 4.a.

The PSC is connected to power and ground, and the clock is set high to allow the inputs to pass through the input registers. Upon execution of the program, it becomes apparent that the output NO is actually the exact inverse of the desired result. An examination of the IC output pads shows that the drivers are composed of three inverters in series; this odd number of inverters results in the inversion of all outputs.

This problem is temporarily corrected by connecting an external inverter to each IC output. With this correction, the logic block error-detection program

is run and no further errors are detected.

The asynchronous control signals -- initial carry-in Co and interrupt acknowledge signal IACK -- are tested as a function of three external inputs $\overline{RES}$, EINT, and IM, and an internal input C1. Co and IACK are found to agree with their respective design equations.

The carry propagation is tested by setting all address bits high and observing the carry-out CO as a function of the carry-in CI. The operation is correct: CO follows the CI signal, with a measured propagation delay of 80 ns, compared to the simulation prediction of about 40 ns (including an output driver delay of 10 ns).

All explicit asynchronous tests have now been executed. Explicit synchronous tests will check the operation of the low and high address bits, C1 and SB. All other circuit functions, including input and output latching, present and return address latching, MUX2 and MUX5 operation, and interrupt handling, will be tested implicitly during the normal operation of the IC.

The next value of the low address bit C1 is a function of its present value and four inputs: $\overline{RES}$, $\overline{INT}$, CS, and $\bar{L}$. When C1 is low, its next value is latched on the falling edge of the square wave clock pulse, as desired. However, when C1 is high, it remains latched high regardless of the inputs. From the equation for C1, it is obvious that when C1 is high, the next value of C1 must always be low. Hence the circuit is not operating correctly.

The layout of the circuit is examined; the C1 section is found to be free of errors. If the circuit has been fabricated correctly, the fault must lie in the actual operation of the designed circuit. The principles of the asynchronous logic have been tested previously; therefore, it is likely that the synchronous logic -- the master-slave data register -- is at fault.

It is hypothesised that the clock signal transitions occur too quickly and produce latching problems due to a faulty register design. The clock transitions are slowed through the circuit of Figure 28. Its operation is very simple: when the CLOCK IN (square wave) is low, the transistor N is off, capacitor C is charged through resistor R, and CLOCK OUT rises; when CLOCK IN is high, C is discharged through N, and CLOCK OUT falls. However, as the CLOCK IN frequency increases, the rise and fall of CLOCK OUT will occupy more and more time relative to the clock period, until finally the signal does not vary from some point between the high and low voltages. This clock degradation will not permit the IC to be tested at very high frequencies.

With the addition of this circuit to the controller system, C1 operates correctly at clock rates of 3.3 MHz and less. At the maximum frequency, R is 10.0 K ohms and C is zero. At 10 KHz, R is 5.7 K ohms and C is 100 microfarads. Decreasing the capacitance decreases the clock's rise and fall times, while varying the resistance serves to position the CLOCK OUT signal at the optimum level.

The next value of the high address bit SB depends on its present value, the latch olock, and four inputs ($\overline{RES}$, $\overline{INT}$, CS, and C1). As for C1, SB only gives the designed results with the addition of the clock circuit of Figure 28. The subroutine bit operates correctly up to a frequency of about 2.4 MHz.

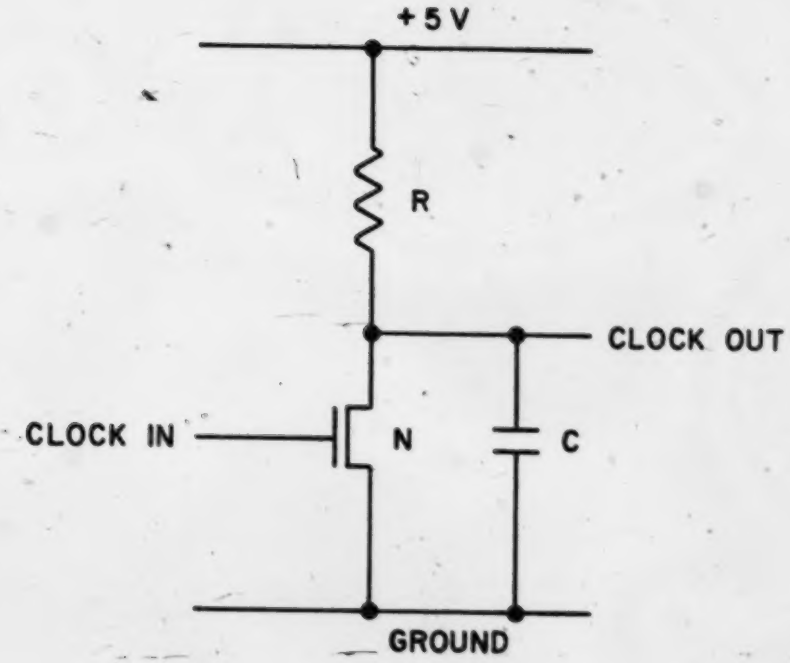


Figure 28. Modified clock circuit to slow rise and fall times.

4.2 Implicit testing

The IC is now connected to three PROMs and is tested under normal operating conditions. The six address bits (SB, A3, A2, A1, A0, and C1) are connected to the lowest six address lines of the PROM chips; all other address lines are grounded. The PROM1 and PROM2 nybbles are connected to the lowest four data lines of two different PROMs; the interrupt mask IM and subroutine call signal CS are connected to the top two bits of the third PROM (the control PROM) data byte.

The various sections of the IC will be tested by storing programs in the three PROMs which will sensitise the circuit to possible errors in those sections.

The increment cycle is tested by running a program which never permits a new address load. For example, all input mask bits may be set high and all input condition bits set low (this test may also be performed explicitly by simply setting the $\overline{L}$ input high). The circuit will then count through the complete address range from 0000 to 1111 and back to 0000 again (SB and C1 remain low). One clock cycle will be required per increment.

The increment test is successfully run at a clock rate of 2.5 MHz. The reset signal $\overline{RES}$ is tested; it causes the address to go to 0000 on the clock cycle immediately after $\overline{RES}$ has gone low, as desired. Counting resumes from 0000 only after $\overline{RES}$ has gone high again.

The load cycle is tested with a program which uses 16 GOTO (unconditional branch) instructions to implement the same four-bit counter (if C1 is included, then this will be a five-bit counter). All bits of each input mask and condition are set high to ensure that $\overline{L}$ will be low; the new address to be loaded will be that of the present address incremented by one. This program will require two cycles per increment, since the second clock cycle (C1 high) is needed to latch

the new address and input and output words.

This test runs successfully at 2.3 MHz. The $\overline{RES}$ line resets the system to 0000 regardless of the current state of the controller (C1 low or high).

Input/output latching is tested by implementing the FSMs of Figure 29. In order to cycle through the FSM of Figure 29.a, the system must be able to latch inputs ranging from 0000 to 0111 and outputs from 0001 to 1000. Similarly, the system realising the FSM of Figure 29.b must latch inputs 1000 to 1111 and outputs 1001 to 0000.

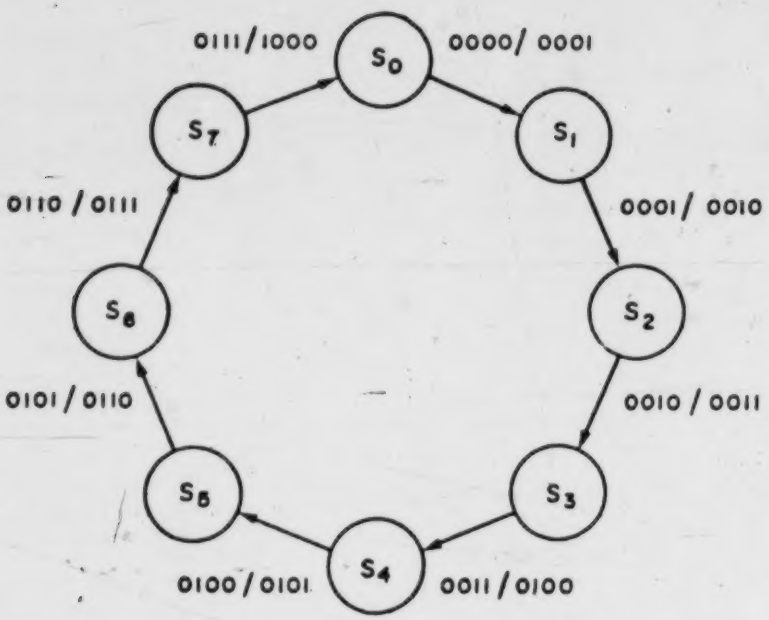
Both of these tests produce the desired results. The outputs (and implicitly the inputs) are passed while C1 and the clock are high, and are held while one of these is low.

Subroutine calls and interrupt handling are tested using the FSM of Figure 30 whose low-level program is shown in Appendix 4.b. The program contains a main program and two subroutines called by the main program; the interrupt vector specifies that the second subroutine will also be used as the interrupt service routine (ISR).

The program runs as specified on the PSC system. Both subroutine and interrupt calls operate correctly, with the SB bit used to change control from main program memory to subroutine program memory. Return from subroutine/ISR is also correct: on return from subroutine, the return address specified during the subroutine call is loaded; on return from interrupt, the address of the instruction at which the interrupt took effect is loaded. This test implicitly confirms the correct operation of the MUX2, which passes either the return address or the present address, and the return address register, which stores the output of MUX2.

All of the above tests confirm the operation of the MUX5, which

a)



b)

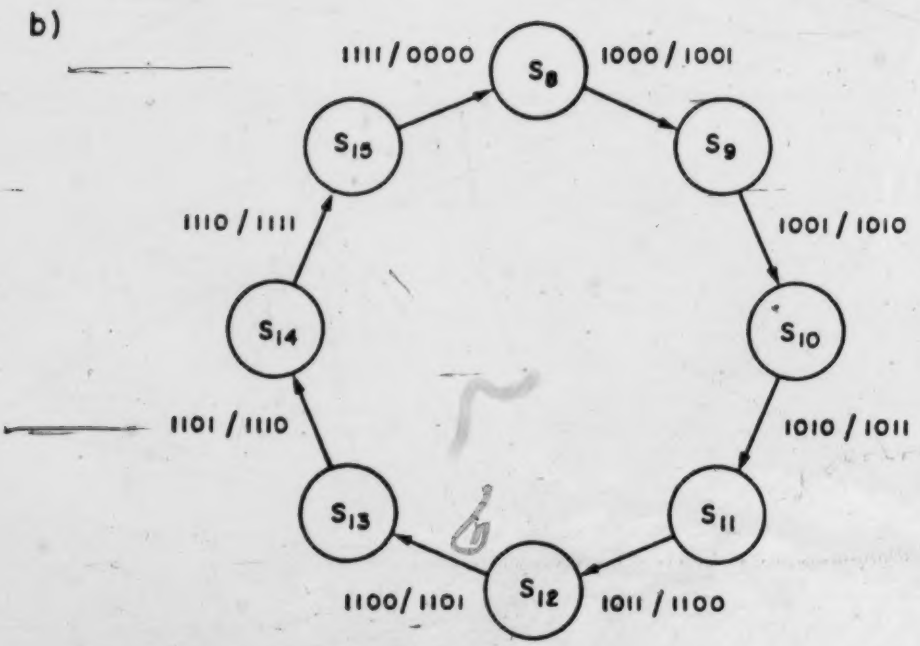


Figure 29. FSMs for I/O latching tests.

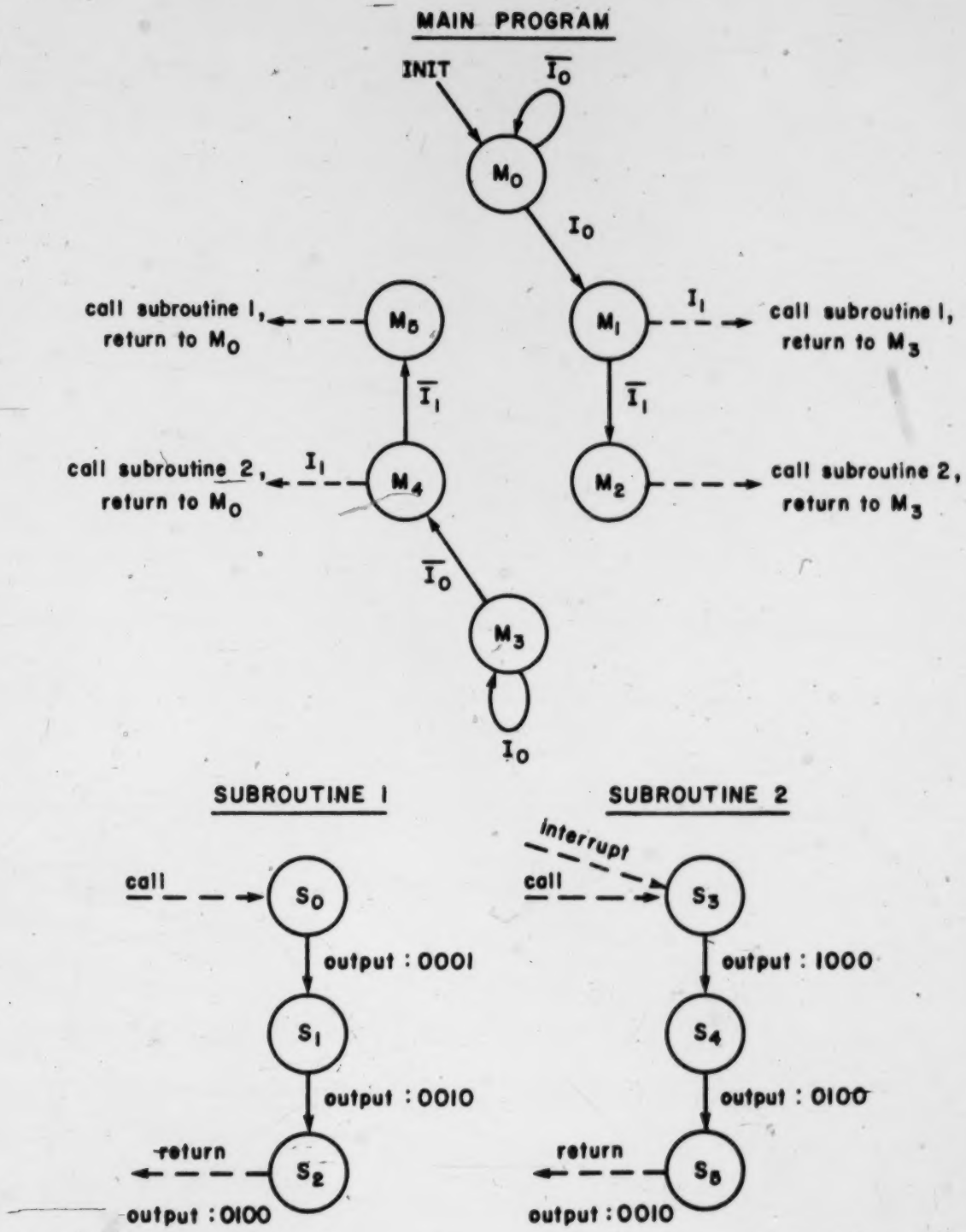


Figure 30. FSM for subroutine/interrupt test.

distinguishes among reset, interrupt, return address, present address, and new address, as well as the correct latching of the MUX5 output by the present address registers (the additional clock circuitry of Figure 28 was necessary to provide proper address latching).

The final test involves the use of the PSC to control a small digital system. The system will calculate the greatest common divisor (GCD) of two numbers.

The GCD of two numbers, A and B, is assumed to be g, where g is nonzero. Then A and B may be written

$$A = (g)(a)$$

$$B = (g)(b) ,$$

where a and b are nonzero. Now B is subtracted from A. If the result is zero, then A equals B and the GCD is A (or B). If the result,

$$C = A - B = (g)(a - b) ,$$

is positive, A can be replaced with C, since C and B have the same GCD as A and B. If C is negative, then B is greater than A, so B is replaced with

$$D = B - A = (g)(b - a) ,$$

since A and D have the same GCD as A and B. The process of subtraction and comparison is iterated. Since A and B are finite and the replacements are always positive and smaller than the original numbers, only a finite number of iterations will be necessary to calculate the GCD.

The FSM which implements this algorithm is shown in Figure 31, in binary decision machine (BDM) form (each state of a BDM has exactly two branches; this will serve to simplify the PSC design of the system [39]). There will be three input lines: an "input ready" signal, and a zero signal and a positive/negative signal which hold the results of a subtraction. Three outputs are

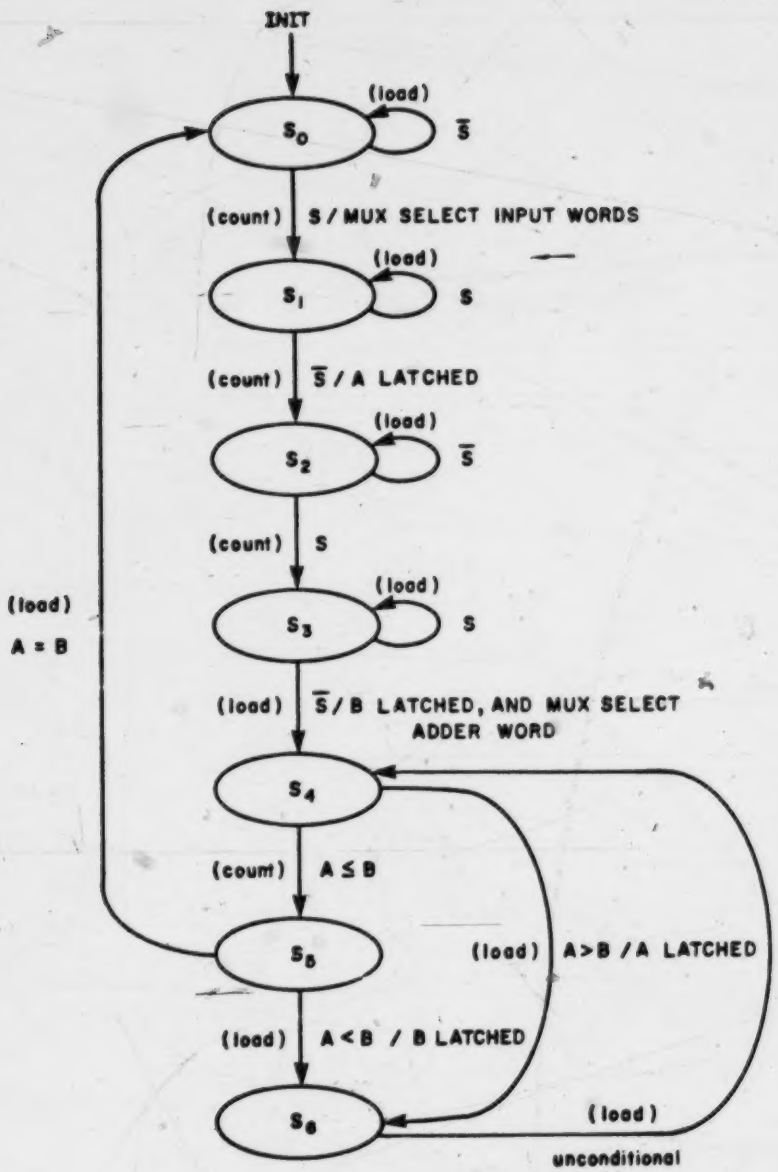


Figure 31. BDM for greatest-common-divisor circuit test.

produced: a multiplexer select line to choose between the original numbers and their replacements, and two register latch lines, to latch a number into one of the two registers.

The BDM contains seven states and fourteen branches. If the PSC program is written directly from the BDM, it will consist of fourteen instructions (70 words). However, some branches do not require an input/output latch. In these cases, all that is needed is a count to access the following instruction. Since this count is performed automatically by the PSC following an unsuccessful branch, an instruction to load the next instruction is not necessary. This is true of the transitions from states S0 to S1, S1 to S2, S2 to S3, S3 to S4, and S5 to S6. The number of instructions necessary will then be only nine, as seen in the PSC low-level language program of Appendix 4.c.

The block diagram of the external circuitry for the system is shown in Figure 32. The PSC inputs and outputs are connected to the external circuit, and the system operates correctly. A GCD requiring n iterations can be calculated in a maximum of $16 + 5(n-1)$ clock cycles.

Since all the signals necessary for cascading two or more chips have been found to operate correctly, the system could easily be expanded to eight or more bits (the necessary connections are shown in Figure 10). Hence, the PSC integrated circuit has been successfully designed to implement control circuits of any size with simple programs running at high speeds.

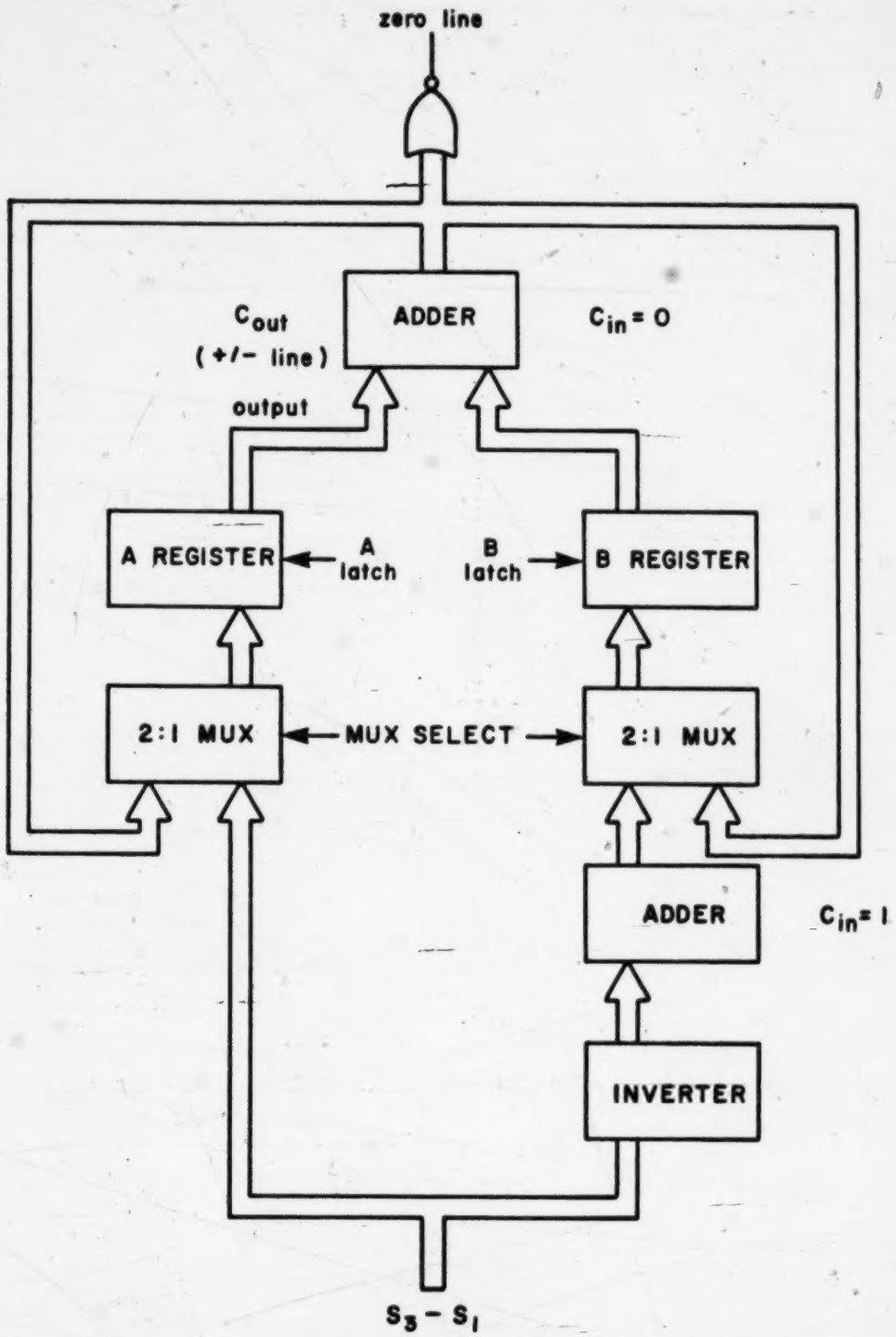


Figure 32. Block diagram of external circuitry for GCD test.

4.3 Timing considerations

Several propagation delays within the IC were measured and compared to the SPICE circuit simulation predictions, as shown in Table 14. The measured delays were consistently longer (two to four times longer) than the predicted delays. For example, the propagation delay of the carry signal was predicted to be about 20 ns without taking into consideration the output driver delay. A circuit simulation of the output driver predicts a worst case propagation delay of eight ns. This results in a total carry delay of about 28 ns, while the actual delay was measured as 82 ns, three times the predicted value.

The discrepancies between measured and predicted delays can be partially attributed to the fact that some signals take longer to rise or fall because they are loaded with several transistor gates (instead of one simple inverter as the load). These gates will hold charge and capacitively slow down the signal transitions. For example, the two-input NAND gate producing Co takes 89 ns to propagate the $\overline{RES}$ signal, instead of the ten ns estimated. This is probably because $\overline{RES}$ is connected to eight gates (sixteen transistors) and is slow to change. Other possible factors are the difference between simulation parameters and the actual parameters of the fabricated circuit, and the fact that the simulated circuit is only an imperfect mathematical model of the actual circuit.

It should be noted that the circuit simulations were meant primarily to give general estimates of the operation of the circuit. In this respect, the simulation results were quite satisfactory.

Using the measured delays, a more accurate estimate of the maximum clock rate can be made. The longest system delay occurs during the load/increment cycle:

INPUT	OUTPUT	PREDICTED DELAY	MEASURED DELAY
INPUT	NO	30	78
MASK	NO	26	70
COND.	NO	22	58
NI	NO	14	28
IM	IACK	12	55
IM	Co	16	85
RESET	Co	10	89
CI	CO	28	82
CK	C1	24	105
CK	SB	24	105

All times in nanoseconds.

Table 14. Comparison of measured and predicted propagation delays within the controller IC.

$$T_{load} = T_{mux5} + T_{co} + T_{adder} + 2(T_{reg}),$$

where the MUX5 delay is now estimated to be about 55 ns, the Co delay is about 30 ns, the adder delay is about 90 ns per chip, and the register delays are estimated to be 10 ns each. This results in a worst case N-chip system delay of

$$T_{system} = 90N + 105,$$

in nanoseconds. For one chip, the maximum clock frequency will be about 5 MHz, quite close to the original estimate of 9 MHz.

This, however, assumes that the access time of the ROM or RAM circuits which hold the FSM program is quite short (under 100 ns). For slower memory chips, the time needed to access the FSM information will have a significant effect on the maximum clock rate.

This also assumes that the master-slave data registers are operating correctly, making the external clock circuitry unnecessary. With this circuitry, however, a conservative maximum clock frequency would be 2.0 MHz, which is still quite satisfactory for most control situations.

4.4 Design modifications

After a circuit has been fully tested, it returns to the design stage of the cycle if any changes are desired or required. The amount of redesign effort needed will depend on the complexity and nature of the correction or addition to be effected.

Three principal elements of the PSC integrated circuit should be redesigned in order to improve the operation of the circuit.

The first and most obvious is the need to produce non-inverting outputs. This can be easily corrected in two outputs -- NO and CO -- by the removal of the inverters immediately preceding the output drivers. The interrupt acknowledge signal IACK can be redefined to be active-high; no modification to the circuit will then be necessary. The remaining eleven outputs will require the addition of minimum-size inverters immediately before the output drivers. This will not present any problems in terms of space and propagation delays.

The second area of redesign involves the data registers and their latch clocks. The master-slave registers for C1, SB, and the present address do not operate reliably when a square wave clock signal is used. However, if the clock transitions are slowed by the circuit of Figure 28, the registers operate satisfactorily.

One possible explanation is that the overlapping of the clock signal CK with its inverse signal $\overline{CK}$ creates latching problems. This could occur if CK went low, closing the master register and opening the slave, while $\overline{CK}$ remained low long enough to allow the slave's output to feed back to the master. An incorrect value might then be stored in the master register.

A SPICE circuit simulation of a master-slave register is performed, using latching characteristics identical to those in the IC. The output of the slave

register is fed back to the input through a two-input NAND gate. The clock pulse is a 12.5 MHz square wave. A plot of the register output versus the clock pulse is shown in Appendix 1.e. The output switches between low and high on the falling edge of the clock, as desired. Similar SPICE runs verify that the present and return address registers, which have a clock signal which is produced by the ANDing of CK and other signals, should also produce the desired results.

A more effective way to locate the problem and produce better register latching is to design, lay out, and fabricate an actual test circuit. This circuit would consist of different register designs, including both single-phase and double-phase clocking mechanisms. Each register would have connections to external inputs and outputs. This would allow the performance of the registers to be measured explicitly, and the most efficient register to be selected for the new modified PSC design.

The third area of the IC design which may be modified is the four-bit half-adder. In the original design, the worst case propagation delay from the carry-in to the carry-out was found to be almost half of the total chip delay, despite the widening of the two-input NANDs and inverters that produce the carry for each bit. For a multi-chip system, the carry delay would occupy and even greater proportion of the total delay. Therefore, the speed of the circuit will be greatly improved by the reduction of the carry delay.

One solution would be to widen the carry-producing gates even further. Circuit simulations predict four-bit worst case propagation delays of 18 nanoseconds for minimum size gates, 14 ns for gates with twice the minimum width, and 12 ns for gates with four times the minimum. However, due to the unreliability of the simulation results, it is not clear whether the actual propagation delay would be greatly improved through the widening of the gates.

Another solution is to employ a simple form of carry look-ahead generation [10]. Carry look-ahead generation involves the production of carries for higher bits before the lower bit sums and carries have stabilised. Figure 33 shows how this could be done for a four-bit half adder.

The sums, S_0 to S_3, are produced by the same half-adder circuits as before, as are the carries C_1 and C_3. However, the intermediate carry C_2, which follows the logic equation

$$C_2 = (CI)(X_0)(X_1),$$

can be produced with a three-input NAND and an inverter. Similarly, the final carry CO (or C_4) has equation

$$CO = (CI)(X_0)(X_1)(X_2)(X_3),$$

which may be formed with a five-input NAND and an inverter. These new circuits will reduce the number of gate levels through which the carries must propagate: C_2 must pass through two levels instead of four, C_3 through four levels instead of six, and CO through two levels instead of eight. Although the gates will sometimes be slower (three- and five-input NANDs have replaced some two-input NANDs), the propagation delays will still be reduced significantly. The additional four gates will occupy a very small amount of silicon area.

The redesigned programmable sequential controller circuit will be faster and more reliable than the original circuit, and will not require the external inverters and clock circuitry.

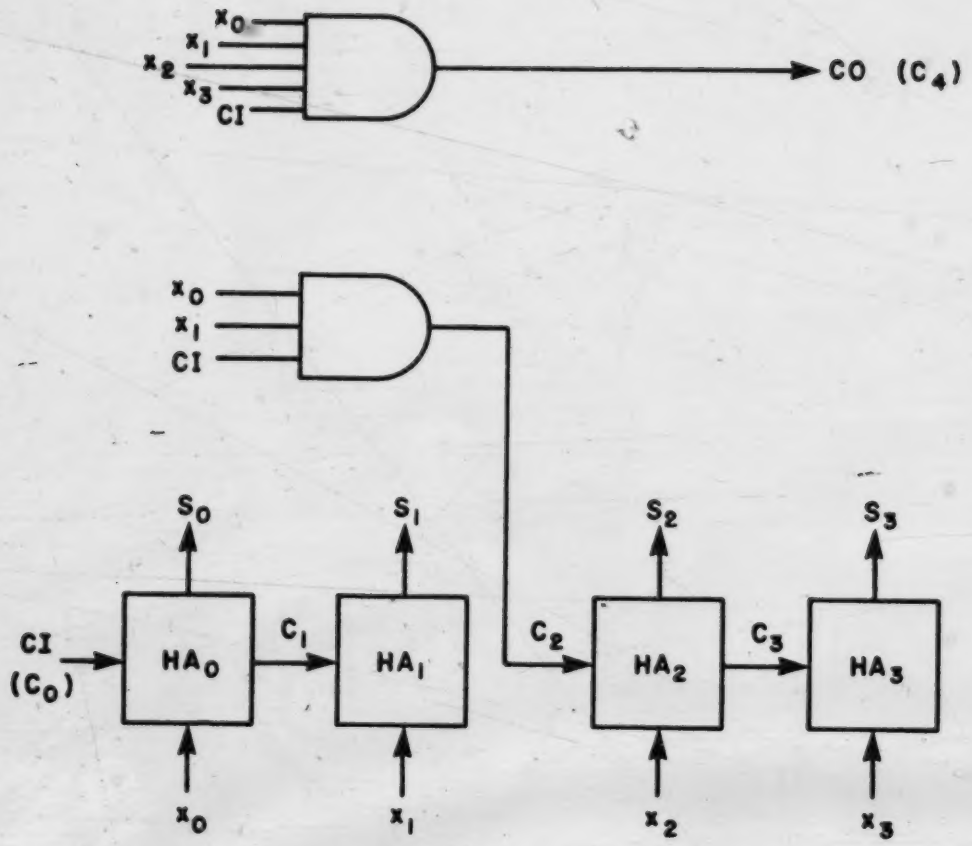


Figure 33. Carry look-ahead generation for four-bit half-adder.

V CONCLUSION

A programmable sequential controller has been designed and constructed to implement any finite-state control situation by examining control inputs and producing state changes and control outputs.

The PSC uses memory segmentation to reduce each branch of a finite state machine to five distinct data fields. This enhances the programmability of the controller and reduces the amount of program memory required to describe the control process.

The controller operates on a semiparallel architecture: logic-related data fields are accessed on the first clock cycle, and if a branch is to occur, output-related data fields are accessed on the second clock cycle. This provides the controller with excellent speed: a count instruction requires only one clock cycle; an n-th priority branch requires n+1 cycles; a subroutine call, two cycles; a return from subroutine, two cycles; and a servicing of an interrupt, three cycles.

The PSC has been implemented as a one-chip integrated circuit. This improves the ease of installation of the system by reducing the size of the circuit and the number and lengths of the interconnections to be made. The propagation delays within the circuit are also greatly improved.

A low-level language has been designed to replace the machine language necessary for system programming. This allows each branch of the state machine to be represented by one simple pseudo-English instruction. An assembler is used to translate this low-level language to the machine language level.

The PSC may be cascaded to control a finite state process of almost unlimited size. A system with n PSC chips will support a process with a maximum of 2^{4n}

main program branches and 2^{4n} subroutine branches. It will require a maximum of $10(2^{4n})$ words of memory; each word will be 4n bits long. The maximum clock frequency will be on the order of $10/(n+1)$ MHz.

The control PROM is used to increase the power of the PSC. Two bits of the control word provide the PSC with subroutine and interrupt control. The remaining bits may be used to provide greater control over input and output ports, programmable timers, and other process-related hardware. For example, one bit may be used to multiplex an input bit or the entire input word, permitting the expansion of the input word without the addition of another PSC chip. Unused bits on the output word may be used to control the input word, in a similar manner.

Thus, an extremely powerful and versatile programmable sequential controller has been designed.

REFERENCES

1 M. S. Abadir and H. K. Reghbati, "LSI Testing Techniques," IEEE Micro, Vol. 3, No. 1, pp. 34-51, 1983.

2 A. Barna and D. Porat, Integrated Circuits in Digital Electronics, New York: John Wiley & Sons, 1983.

3 G. Billingsley and K. Keller, "KIC: A graphics editor for integrated circuits," University of California at Berkeley.

4 T. R. Blakeslee, Digital Design with Standard MSI and LSI, New York: John Wiley & Sons, 1975.

5 S. J. Cahill, Designing Microprocessor-based Digital Circuitry, Englewood Cliffs, N.J.: Prentice-Hall, 1985.

6 S. J. Cahill, Digital and Microprocessor Engineering, Chichester, England: Ellis Horwood, 1982.

7 M. P. Chinitz, The Logic Design of Computers, Indianapolis: Howard W. Sams, 1981.

8 D. M. Divan, G. C. Hancock, G. S. Hope, and T. H. Barton, "Micro-programmable sequential controller," IEE Proc., 6, pp. 201-208, 1984.

9 W. Fletcher, An Engineering Approach to Digital Design, Englewood Cliffs, N.J.: Prentice-Hall, 1980.

10 J. B. Gosling, Design of Arithmetic Units for Digital Computers, New York: Springer-Verlag, 1980.

11 P. R. Gray and R. G. Meyer, Analysis and Design of Analog Integrated Circuits, New York: John Wiley & Sons, 1984.

12 V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, Computer Organization, New York: McGraw-Hill, 1984.

13 A. B. Hayes, "Stored state asynchronous sequential circuits," IEEE T-C, 8, pp. 596-600, 1981.

14 G. S. Hope, Integrated Devices in Digital Circuit Design, New York: John Wiley & Sons, 1981.

15 M. J. Howes and D. V. Morgan (Editors): Large Scale Integration (Devices, Circuits, and Systems), New York: John Wiley & Sons, 1981.

16 R. Kenny, "Microprogramming simplifies control system design," Comput. Des., 14, pp. 96-98,1975.

17 R. H. Krambeck, C. M. Lee, and H. S. Law, "High-speed compact circuits with CMOS," IEEE J. of SSC, 3, pp. 614-619, 1982.

18 L. A. Leventhal, 6809 Assembly Language Programming, Berkeley: Osborne/ McGraw-Hill, 1981.

19 L. F. Lind, J. C. C. Nelson, Analysis and Design of Sequential Digital Systems, New York: John Wiley & Sons, 1977.

20 P. F. Lister, Single-chip Microcomputers, London: Granada, 1984.

21 J. B. Maginnis, Elements of Compiler Construction, New York: Appleton-Century-Crofts, 1972.

22 D. A. Mange, "A High-Level-Language Programmable Controller," IEEE Micro, Vol. 6, No. 1, pp. 25-41, 1986.

23 M. M. Mano, Digital Logic and Computer Design, Englewood Cliffs, N.J.: Prentice-Hall, 1979.

24 J. Mavor, M. A. Jack, and P. B. Denyer, Introduction to MOS LSI Design, London: Addison-Wesley, 1983.

25 J. Mavor (Editor), M.O.S.T. Integrated Circuit Engineering, Stevenage, England: Peter Peregrinus, 1973.

26 F. F. Mazda, Integrated Circuits (Technology and Applications), Cambridge; New York: Cambridge University Press, 1978.

27 D. R. McGlynn, Fundamentals of Microcomputer Programming Including Pascal, New York: John Wiley and Sons, 1982.

28 D. R. McGlynn, Modern Microprocessor System Design: Sixteen-bit and Bit-slice Architecture, New York: John Wiley & Sons, 1980.

29 C. Mead and L. Conway, Introduction to VLSI systems, Reading, Mass.: Addison-Wesley, 1980.

30 G. Musgrave (Editor), Computer-Aided Design of Digital Electronic Circuits and Systems, Amsterdam: North-Holland, 1979.

31 G. J. Myers, Digital System Design with LSI Bit-Slice Logic, New York: John Wiley and Sons, 1980.

32 S. S. Patil and T. A. Welch, "Programmable logic approach for VLSI," IEEE T-C, Vol C-28, No. 9, pp. 594-601, 1979.

33 B. Randell and P. C. Treleaven (Editors), VLSI Architecture, Englewood Cliffs, N.J.: Prentice Hall, 1983.

34 C. H. Roth, Fundamentals of Logic Design, St. Paul, Minn.: West, 1979.

35 H. S. Stone, Microcomputer Interfacing, Reading, Mass: Addison-Wesley, 1982.

36 J. P. Tremblay and R. Manohar, Discrete Mathematical Structures with Applications to Computer Science, New York: McGraw-Hill, 1975.

37 A. Vladimirescu et al., "SPICE user's guide," University of California at Berkeley, 1981.

38 C. A. Wiatrowski and C. H. House, Logic Circuits and Microcomputer Systems, New York: McGraw-Hill, 1980.

39 P. J. A. Zsombor-Murray, L. J. Vroomen, R. D. Hudson, T. Le-Ngoc, and P. H. Holck, "Binary-Decision-Based Programmable Controllers," IEEE Micro, Vol. 3, No. 4, pp. 67-83, 1983.

40 "A set of simplified CMOS layout rules," Kingston: Canadian Microelectronics Corporation, 1984.

APPENDIX 1. SPICE circuit simulations.

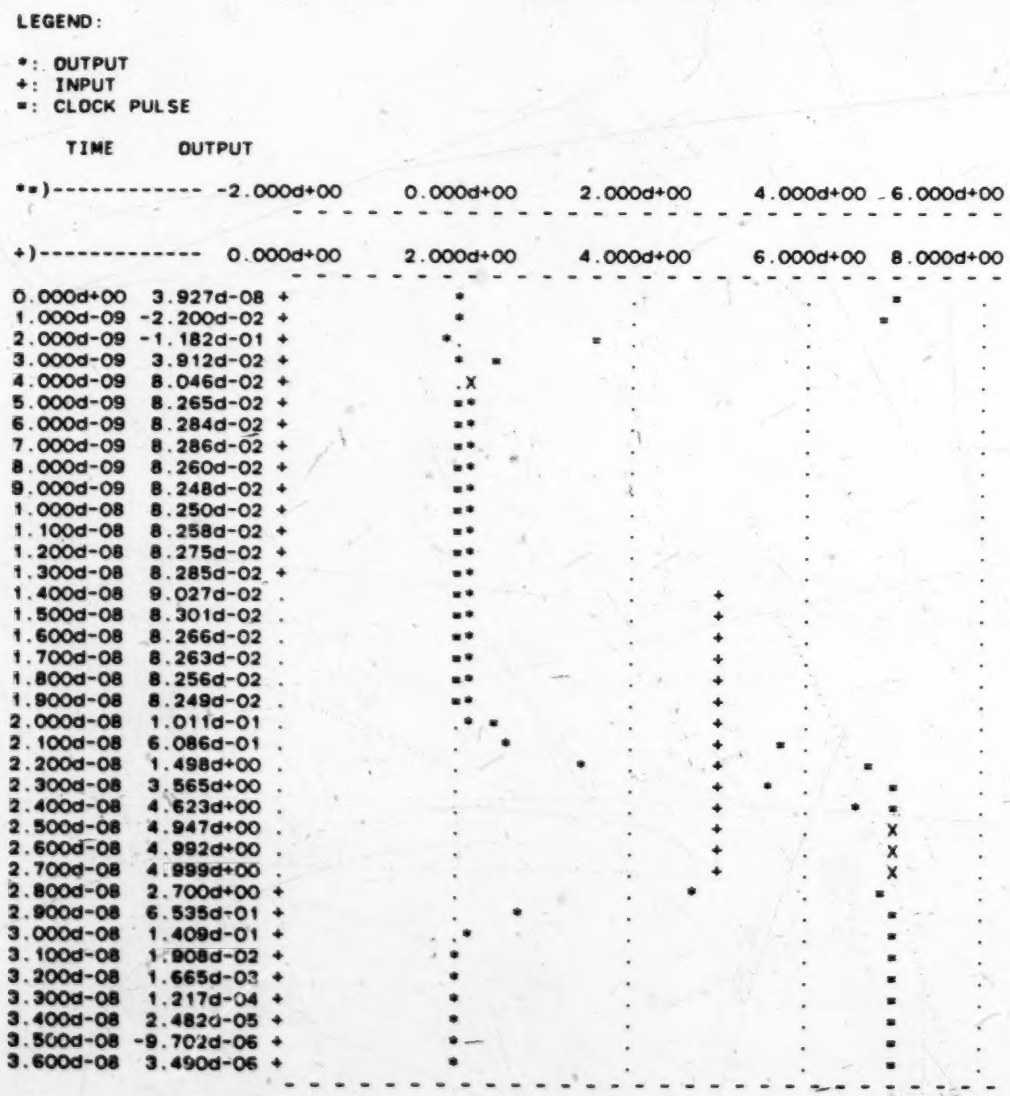
a) Data register simulation.

```
*******06/05/85 ********  SPICE 2G.6    3/15/83 ********10:34:17*****

REGISTER LOADING TEST TO DETERMINE OPERATION AND DELAYS:

****      INPUT LISTING                    TEMPERATURE =   27.000 DEG C

******************************************************************************


.MODEL PDEV PMOS (LEVEL=2 PB=0.7 CGSO=2.44E-10 CGDO=2.44E-10
+CGBO=2E-12 RSH=75.0 CJ=1.54E-4 MJ=0.5 CJSW=4.37E-10
+MJSW=0.5 JS=4.19E-10 TOX=8.5E-8 NSUB=1.98E15 TPG=1.0
+XJ=0.9E-6 LD=0.6E-6 UO=240 UCRIT=6.44E4 UEXP=0.139
+VMAX=7.33E4 XQC=0.4 VTO=-0.9 KP=9.75E-6 GAMMA=0.634 PHI=0.612
+LAMBDA=3.0E-2 RD=2.0 RS=2.0 CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14)

.MODEL NDEV NMOS (LEVEL=2 PB=0.7 CGSO=2.84E-10 CGDO=2.84E-10
+CGBO=2E-12 RSH=15.0 CJ=3.44E-4 MJ=0.5 CJSW=1.09E-9
+MJSW=0.5 JS=1.37E-5 TOX=8.5E-8 NSUB=9.92E15 TPG=1.0
+XJ=1.0E-6 LD=0.7E-6 UO=750 UCRIT=1.23E5 UEXP=0.022
+VMAX=4.92E5 XQC=0.4 VTO=0.9 KP=3.05E-5 GAMMA=1.592 PHI=0.695
+LAMBDA=1.0E-2 RD=2.0 RS=2.0 CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14)

VDD 10 0 DC 5
.OPT NOMOD ABSTOL=100PA VNTOL=100UV NUMDGT=3
.WIDTH OUT=80

*INVERTER SUBCIRCUIT:
.SUBCKT INV 1 2 3
MS1 1 2 3 3 PDEV L=5U W=16U
MS2 1 2 0 0 NDEV L=5U W=5U
.ENDS

*GATE SUBCIRCUIT:
.SUBCKT GATE 1 2 3 4 5
MS3 1 3 2 0 NDEV L=5U W=5U
MS4 1 4 2 5 PDEV L=5U W=16U
.ENDS

*LOADING TEST FOR REGISTER:
XGATE1 4 1 2 3 10 GATE
XINV1 3 2 10 INV
XINV2 5 4 10 INV
XINV3 6 5 10 INV
XGATE2 4 6 3 7 10 GATE
XINV4 7 3 10 INV
*
*VIN IS INPUT SIGNAL; VLAT IS CLOCK PULSE:
VIN 1 0 PULSE (0.0 5.0 10.5N 0.0 0.0 15.0N 30.0N)
VLAT 2 0 PULSE (0.0 5.0 5.0N 0.0 0.0 5.0N 15.0N)
.TRAN 2.0NS 44.0NS
* OBTAIN PLOT OF OUTPUT, INPUT, AND CLOCK SIGNALS:
.PLOT TRAN V(4) V(1) V(2)
.END
```

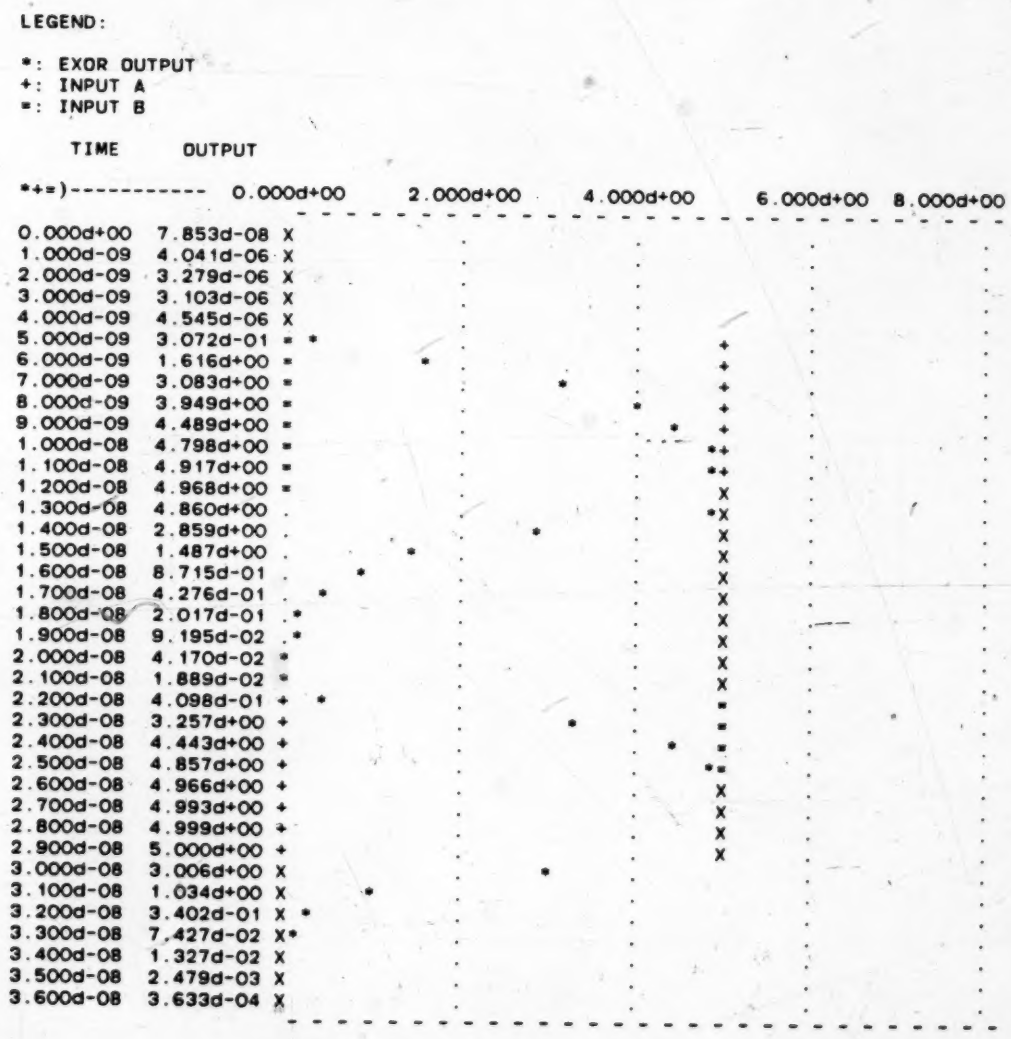
```
*******06/05/85 ********  SPICE 2G.6    3/15/83 ********10:34:17*****

REGISTER LOADING TEST TO DETERMINE OPERATION AND DELAYS:

****      INITIAL TRANSIENT SOLUTION       TEMPERATURE =   27.000 DEG C

******************************************************************************

 NODE   VOLTAGE     NODE   VOLTAGE     NODE   VOLTAGE     NODE   VOLTAGE

(  1)     .0000    (  2)     .0000    (  3)    5.0000    (  4)     .0000

(  5)    5.0000    (  6)     .0000    (  7)     .0000    ( 10)    5.0000

    VOLTAGE SOURCE CURRENTS

    NAME        CURRENT

    VDD       -5.547e-11

    VIN        6.933e-12

    VLAT       0.000e+00

    TOTAL POWER DISSIPATION   2.77e-10  WATTS
*******06/05/85 ********  SPICE 2G.6    3/15/83 ********10:34:17*****

REGISTER LOADING TEST TO DETERMINE OPERATION AND DELAYS:

****      OPERATING POINT INFORMATION      TEMPERATURE =   27.000 DEG C

******************************************************************************

**** MOSFETS

             MS3.XGA*  MS4.XGA*  MS1.XIN*  MS2.XIN*  MS1.XIN*  MS2.XIN*  MS1.XIN*

MODEL      NDEV      PDEV      PDEV      NDEV      PDEV      NDEV      PDEV
ID         2.69e-17 -1.93e-12 -6.20e-12  1.93e-12 -6.94e-12  1.93e-12 -1.93e-12
VGS            .000      5.000     -5.000      .000     -5.000      .000      .000
VDS            .000      .000      .000      5.000      .000      5.000     -5.000
VBS            .000      5.000      .000      .000      .000      .000      .000

             MS2.XIN*  MS3.XGA*  MS4.XGA*  MS1.XIN*  MS2.XIN*

MODEL      NDEV      NDEV      PDEV      PDEV      NDEV
ID         2.77e-11  1.39e-11 -1.93e-12 -1.93e-12  6.81e-12
VGS           5.000     5.000      .000      .000     5.000
VDS            .000      .000      .000     -5.000      .000
VBS            .000      .000     5.000      .000      .000
```

```
*******06/05/85 ********  SPICE 2G.6    3/15/83 ********10:34:17*****

REGISTER LOADING TEST TO DETERMINE OPERATION AND DELAYS:

****     TRANSIENT ANALYSIS                 TEMPERATURE =   27.000 DEG C

******************************************************************************

LEGEND:

*: V(4)
+: V(1)
=: V(2)

    TIME       V(4)

*)------------- -2.000e+00     0.000e+00     2.000e+00     4.000e+00  6.000e+00
                - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

+=)------------  0.000e+00     2.000e+00     4.000e+00     6.000e+00  8.000e+00
                - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0.000e+00   2.356e-07 X          *                .             .          .
2.000e-09   9.397e-05 X          *                .             .          .
4.000e-09   2.014e-04 X          *                .             .          .
6.000e-09   7.848e-02 +           *               .             =          .
8.000e-09   7.670e-03 +          *                .             =          .
1.000e-08   1.172e-03 +          *                .             =          .
1.200e-08   2.416e+00 =          .                *             +          .
1.400e-08   2.087e+00 =          .                .*            +          .
1.600e-08   7.485e-01 =          .                .  *          +          .
1.800e-08   2.136e-01 =          .                .*            +          .
2.000e-08   5.376e-02 =          *                .             +          .
2.200e-08   1.170e+00 .          .                .             X          .
2.400e-08   4.099e+00 .          .                .             X           *
2.600e-08   4.900e+00 .          .                .             X              *
2.800e-08   4.058e+00 X          .                .                         .
3.000e-08   4.056e+00 X          .                .                         .
3.200e-08   4.655e+00 X          .                .                          *
3.400e-08   4.889e+00 X          .                .                           *
3.600e-08   4.429e+00 +          .                .             =           .
3.800e-08   1.657e+00 +          .                .           *=            .
4.000e-08   1.986e-01 +          *                .             =           .
4.200e-08   2.494e+00 =          .                .*            +          .
4.400e-08   2.165e+00 =          .                .  *          +          .
                - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        JOB CONCLUDED

         TOTAL JOB TIME            189.05
```

b) SPICE plot of transmission gate operation.

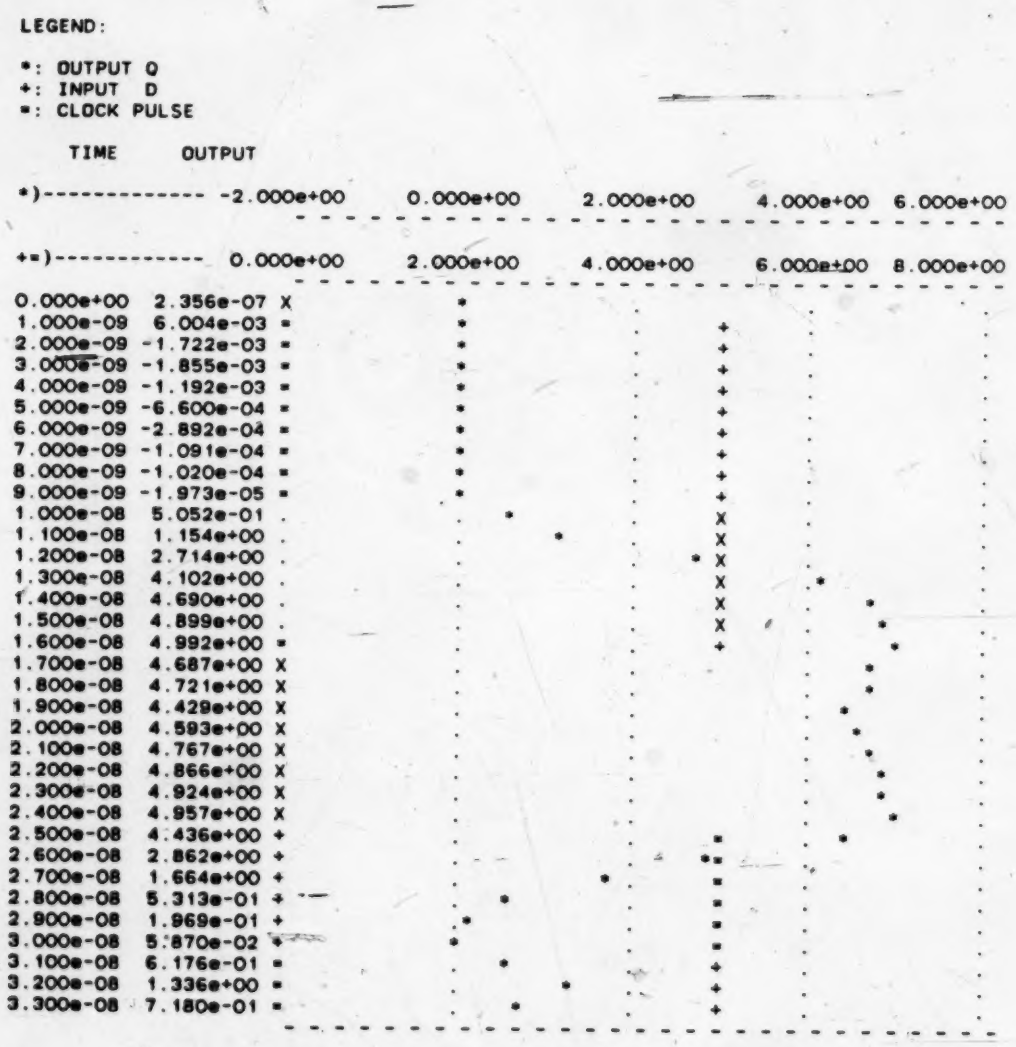


c) SPICE plot of exclusive-or gate operation.



```
LEGEND:

*: EXOR OUTPUT
+: INPUT A
=: INPUT B

    TIME       OUTPUT

*+=)---------- 0.000d+00      2.000d+00      4.000d+00      6.000d+00  8.000d+00
0.000d+00   7.853d-08 X
1.000d-09   4.041d-06 X
2.000d-09   3.279d-06 X
3.000d-09   3.103d-06 X
4.000d-09   4.545d-06 X
5.000d-09   3.072d-01 = *
6.000d-09   1.616d+00 =   *
7.000d-09   3.083d+00 =     *
8.000d-09   3.949d+00 =       *
9.000d-09   4.489d+00 =         *
1.000d-08   4.798d+00 =           *+
1.100d-08   4.917d+00 =           *+
1.200d-08   4.968d+00 =            X
1.300d-08   4.860d+00 .           *X
1.400d-08   2.859d+00 .     *      X
1.500d-08   1.487d+00 .   *        X
1.600d-08   8.715d-01 .  *         X
1.700d-08   4.276d-01 . *          X
1.800d-08   2.017d-01 .*           X
1.900d-08   9.195d-02 .*           X
2.000d-08   4.170d-02 *            X
2.100d-08   1.889d-02 *            X
2.200d-08   4.098d-01 +  *         =
2.300d-08   3.257d+00 +    *       =
2.400d-08   4.443d+00 +      *     =
2.500d-08   4.857d+00 +        *   =
2.600d-08   4.966d+00 +            X
2.700d-08   4.993d+00 +            X
2.800d-08   4.999d+00 +            X
2.900d-08   5.000d+00 +            X
3.000d-08   3.006d+00 X     *
3.100d-08   1.034d+00 X   *
3.200d-08   3.402d-01 X *
3.300d-08   7.427d-02 X*
3.400d-08   1.327d-02 X
3.500d-08   2.479d-03 X
3.600d-08   3.633d-04 X
```

d) SPICE plot of data register operation.



e) SPICE plot of master-slave register operation.

```
LEGEND:

*: MASTER-SLAVE REGISTER OUTPUT
+: CLOCK PULSE

     TIME      OUTPUT

*)------------- -2.000d+00     0.000d+00     2.000d+00     4.000d+00  6.000d+00
                - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

+)------------- 0.000d+00      2.000d+00     4.000d+00     6.000d+00  8.000d+00
                - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0.000d+00   2.609d+00 +                                .
1.000d-08   3.701d+00 .                                .
2.000d-08   4.983d+00 .                                .
3.000d-08   4.999d+00 .                                .
4.000d-08   5.000d+00 .                                .
5.000d-08   5.000d+00 .                                .
6.000d-08   2.637d+00 +                                .
7.000d-08   3.044d-01 +                                .
8.000d-08   2.009d-02 +                                .
9.000d-08   1.331d-02 .                                .
1.000d-07   7.175d-03 .                                .
1.100d-07   5.550d-04 .                                .
1.200d-07  -6.184d-05 .                                .
1.300d-07   1.219d-04 .                                .
1.400d-07   3.043d+00 +                                .
1.500d-07   4.776d+00 +                                .
1.600d-07   4.979d+00 +                                .
1.700d-07   4.795d+00 .                                .
1.800d-07   4.995d+00 .                                .
1.900d-07   5.000d+00 .                                .
2.000d-07   5.000d+00 .                                .
2.100d-07   5.000d+00 .                                .
2.200d-07   2.634d+00 +                                .
2.300d-07   3.038d-01 +                                .
2.400d-07   2.003d-02 +                                .
2.500d-07   2.175d-02 .                                .
2.600d-07   7.217d-03 .                                .
2.700d-07   6.134d-04 .                                .
2.800d-07   3.840d-05 .                                .
2.900d-07   1.217d-05 .                                .
3.000d-07   3.034d+00 +                                .
                - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

APPENDIX 2. PSC low-level language assembler program.

```
begin
comment

AUTHOR:          WERNER E FRAGA
DATE:            20 JULY 1985
PROGRAM NAME:    PSC.S
PURPOSE:         THIS PROGRAM IS  A COMPILER FOR THE PSC LANGUAGE.  IT
          TAKES A LOW LEVEL LANGUAGE AS INPUT  AND PRODUCES AN OUTPUT
          FILE IN  MC6809 ASSEMBLER  WHICH CAN BE CROSSASSEMBLED INTO
          HEXADECIMAL CODE  TO BE LOADED INTO THREE EPROMS TO BE USED
          IN CONNECTION  WITH THE  PROGRAMMABLE SEQUENTIAL CONTROLLER
          (PSC) CHIP DESIGNED BY THE AUTHOR AND FABRICATED BY THE CA-
          NADIAM MICROELECTRONICS CORPORATION (CMC).

COMPILATION:     TO COMPILE ON MTS, THE COMMAND

          $RUN *ALGOLW SCARDS=PSC.S SPUNCH=OBJECT SPRINT=LIST

          TAKES THIS PROGRAM  (IN 'PSC.S')  AND PLACES THE EXECUTABLE
          CODE IN 'OBJECT' AND A LISTING OF THE PROGRAM IN 'LIST'.

RUNNING:         TO RUN THE COMPILED PROGRAM ON MTS, THE COMMAND

          $RUN OBJECT 1=PSC.IN 2=PROM1 3=PROM2 4=CONTROLPROM ...
          ... 5=ERRORMSGS T=1

          TAKES THE LOW-LEVEL LANGUAGE INPUT ('PSC.IN')  AND CONVERTS
          IT TO MC6809 ASSEMBLER,  PLACING THE RESULT IN THREE OUTPUT
          FILES:  'PROM1' CONTAINS THE INPUT MASK AND OUTPUT INFORMA-
          TION, 'PROM2' CONTAINS THE INPUT CONDITION AND NEXT ADDRESS
          INFORMATION,  AND 'CONTROLPROM' CONTAINS THE INTERRUPT MASK
          BIT, SUBROUTINE CALL BIT, AND I/O CONTROL BITS.  THE FOURTH
          OUTPUT FILE, 'ERRORMSGS', CONTAINS THE ERROR MESSAGES GENE-
          RATED BY THIS PROGRAM.  IF ANY MESSAGES ARE GENERATED, THEN
          THERE IS AN ERROR  IN THE INPUT FILE,  AND THE  OUTPUT PRO-
          DUCED IS NOT VALID.

          IF NO ERRORS ARE FOUND,  THEN THE THREE OUTPUT FILES MAY BE
          BE CROSSASSEMBLED ON MTS USING THE COMMAND

          $RUN UNSP:M6809ASM SCARDS=6809FILE SPUNCH=XASM

          WHERE  '6809FILE'  IS ONE  OF THE THREE  OUTPUT  FILES  AND
          'XASM' IS THE HEXADECIMAL OUTPUT FILE, WHICH IS READY TO BE
          LOADED INTO A PROM TO BE USED AS PART OF THE PSC SYSTEM BE-
          ING DEVELOPED.

INPUT SPECS:     THE LANGUAGE CONSISTS OF COMBINATIONS OF NINE COMMAND
          TYPES,  EACH OCCUPYING A SINGLE LINE  IN THE INPUT FILE AND
          STARTING IN THE FIRST COLUMN.  THESE COMMANDS ARE:

     1    NODExxx  -- SPECIFIES  A NEW NODE  IN THE  FINITE STATE MA-
          CHINE.   THE NODE IS UNIQUELY DEFINED  BY THE THREE CHARAC-
          TERS 'xxx' IMMEDIATELY FOLLOWING 'NODE'.  'x' IS ANY ALPHA-
          NUMERIC CHARACTER.  A NODE MAY NOT BE SPECIFIED  MORE THAN
          THAN ONCE IN EITHER THE MAIN ROUTINE OR SUBROUTINE SECTION.
          COMMANDS DESCRIBING THE NODE ARE PLACED ON FOLLOWING LINES,
          UNTIL ANOTHER NODE IS SPECIFIED.

     2    IF 11111111 NODExxx[, OUT 00000000]  -- SPECIFIES A  CONDI-
```

TIONAL BRANCH. THE INPUT '11...' IS AN EIGHT-BIT STRING OF 1'S, 0'S, AND X'S (DON'T CARES). UPON THE RECEIPT OF SUCH AN INPUT, THE PSC IS TO GO TO THE NODE SPECIFIED BY THE THREE CHARACTER STRING 'xxx' WHICH MUST HAVE BEEN SPECIFIED ELSEWHERE IN THE INPUT FILE. AN OUTPUT MAY BE SPECIFIED BY FOLLOWING THE 'xxx' WITH A ',' AND THE LABEL 'OUT'. THE OUTPUT STRING 'oo...' MUST CONSIST OF EIGHT 1'S AND 0'S (THE '[...]' IMPLIES 'OPTIONAL').

3 GOTO NODExxx[, OUT oooooooo] -- SPECIFIES AN UNCONDITIONAL BRANCH FROM THE PRESENT NODE TO THE NODE 'xxx' SPECIFIED ELSEWHERE. THE OPTIONAL OUTPUT IS AS FOR THE CONDITIONAL BRANCH.

4 SUBROUTINES -- SPECIFIES THE ROUTINE SECTION OF THE FSM HAS ENDED AND THE SUBROUTINE SECTION BEGINS. THIS COMMAND HAS NO PARAMETERS.

5 JUMP NODExxx, RETURN NODEyyy[, OUT oooooooo] -- SPECIFIES A CALL OF THE SUBROUTINE WHICH STARTS AT NODE 'xxx'. THE OPTIONAL OUTPUT IS SPECIFIED AS FOR THE BRANCHES ABOVE. ALL 'JUMP' COMMANDS MUST LIE WITHIN THE ROUTINE SECTION, MUST CALL A SUBROUTINE LYING IN THE SUBROUTINE SECTION, AND MUST RETURN TO THE ROUTINE SECTION(I.E. NODE 'xxx' MUST BE SPECIFIED AFTER THE 'SUBROUTINES' COMMAND, WHILE NODE 'yyy' MUST BE SPECIFIED BEFORE THE 'SUBROUTINES' COMMAND.

6 RTS[,OUT oooooooo] -- SPECIFIES A RETURN FROM SUBROUTINE TO THE NODE SPECIFIED IN THE 'JUMP' COMMAND. THIS COMMAND MUST LIE IN THE SUBROUTINE SECTION. OPTIONAL OUTPUT IS AS BEFORE.

7 MASK -- SETS THE INTERRUPT MASK IN THE CONTROL PROM TO PREVENT INTERRUPTS FROM OCCURRING WHILE THE PSC IS IN THE PRESENT NODE. THE MASK COMMAND HAS NO PARAMETERS.

8 VECTOR INTERRUPTS NODExxx[, OUT oooooooo] -- SPECIFIES THE INTERRUPT VECTOR AS NODE 'xxx'. OPTIONAL OUTPUT AS ABOVE. THE NODE SET AS THE VECTOR MUST BE SPECIFIED IN THE SUBROUTINE SECTION, AS THE INTERRUPT IS TO BE SERVICED BY A SUBROUTINE, WITH RETURN TO THE ROUTINE SECTION BY 'RTS'. THE 'VECTOR...' COMMAND MUST ALSO LIE IN THE SUBROUTINE SECTION.

9 END -- SPECIFIES THE END OF THE INPUT FILE. THE END COMMAND HAS NO PARAMETERS.

THE INPUT FILE MUST BEGIN WITH THE COMMAND 'NODE..' AND END WITH THE COMMAND 'END'. THE MAXIMUM NUMBER OF BRANCH INSTRUCTIONS IS 256 FOR EACH SECTION.

OUTPUT SPECS: THE THREE OUTPUT FILES CONSIST OF MC6809 ASSEMBLER INSTRUCTIONS. ONLY THREE TYPES OF INSTRUCTIONS ARE USED:

1 ORG aaa -- SPECIFIES THE STARTING ADDRESS OF THE BYTES OF CODE WHICH FOLLOW.

2 FCB ccc -- (FORM CONSTANT BYTE) SPECIFIES A BYTE OF INFORMATION.

```
        3  END  --  SIGNIFIES THE END OF THE MC6809 PROGRAM.

           FILE PROM1 CONTAINS TWO BYTES OF INFORMATION PER BRANCH IN-
           STRUCTION.  THE FIRST IS THE INPUT MASK.  THE SECOND IS THE
           OUTPUT.
           FILE PROM2 CONTAINS TWO BYTES OF INFORMATION PER BRANCH IN-
           STRUCTION. THE FIRST IS THE INPUT CONDITION.  THE SECOND IS
           THE NEXT ADDRESS OR STATE.
           FILE CONTROLPROM  CONTAINS ONE BYTE  OF INFORMATION PER IN-
           STRUCTION.  IT SPECIFIES THE INTERRUPT MASK BIT, SUBROUTINE
           CALL BIT, AND I/O CONTROL BITS.

PROGRAM DESCRIPTION:
        THE PROGRAM IS COMPOSED OF TWO PARTS.  THE FIRST PART CON-
        SISTS OF SEARCHING  THE INPUT FILE FOR 'NODE' COMMANDS  AND
        ASSIGNING EACH UNIQUE NODE AN ADDRESS IN THE PROM ACCORDING
        TO HOW MANY BRANCH INSTRUCTIONS FOLLOW THAT 'NODE' COMMAND.
        THE SECOND PART  INVOLVES  THE TRANSLATION  OF EACH COMMAND
        INTO MC6809 ASSEMBLER  USING THE ADDRESSES OBTAINED  IN THE
        FIRST PART.   EACH PROCEDURE IS DOCUMENTED  WITHIN THE PRO-
        GRAM ITSELF.

CALLING HIERARCHY:  DESCRIPTION OF PROCEDURE CALLING LEVELS...

        MAIN ---->>  INITIALISE
             ---->>  GETADDRESS  ---->>  GETNODE
             ---->>  TRANSLATE   ---->>  IFBRANCH
                                 ---->>  BRANCH     ----> FINDNODE
                                 ---->>  JUMP       ----> OUT
                                 ---->>  RTS
                                 ---->>  SERVICE
                                 ---->>  SBRT
                                 ---->>  MASK
                                 ---->>  CONTROL
                                 ---->>  FINISH
             ---->>  SETSERV

-------------------------------------------------------------------------

GLOBAL VARIABLES:

  string(3) array order(1::512);
  string(8) int_out;
  integer array address(1::512);
  integer count, servadd, line;
  logical subflag, maskbit, subbit;

comment  PROCEDURE  INITIALISE  SETS  ALL GLOBAL LOGICALS  TO FALSE,
         OUTPUTS THE HEADERS FOR THE THREE PROM ASSEMBLER FILES, AND
         CLEARS THE INTERRUPT SERVICE VECTOR.  IT IS CALLED BY MAIN;

  procedure initialise;
    begin
      subbit:=maskbit:=subflag:= false;
      put(2,"A6"," ORG 0");
      put(3,"A6"," ORG 0");
      put(4,"A6"," ORG 0");
      servadd:=0;
      int_out:="00000000";
    end;
```

```
comment  PROCEDURE  GETADDRESS  FINDS LINES  STARTING WITH  'N'  AND
         CALLS GETNODE TO STORE THE NODE. IT THEN ASSIGNS AN ADDRESS
         TO THE NODE  AND READS THE INPUT FILE  UNTIL ANOTHER 'N' IS
         FOUND.   EACH TIME A COMMAND STARTING WITH  'I','G','J', OR
         'R' IS FOUND,  THE ADDRESS OF THE NEXT NODE  IS INCREMENTED
         BY ONE TO ALLOW  FOR THE BRANCH COMMANDS TO FIT IN THE OUT-
         PUT FILES.  GETADDRESS IS CALLED BY MAIN;

  procedure getaddress;
    begin
      logical done;
      string(1) char;
      string(10) sub;
      done:=false;
      line:= 0;
      count:=1;
      address(1):= address(257):= 0;
      while ¬done do
        begin
          line:=line+1;
          get(1,"A1",char);
          if char = "N" then
            begin
              getnode;
              address(count):=address(count-1);
            end
          else if char="I" or char="G" or char="J" or char="R"
            then address(count):= address(count)  + 1
            else if char="S" then count:=257
              else if char="E" then done:=true
                else if char¬="V" and char¬="M" then
                  put(5,"A40","ERROR: BAD FIRST LETTER AT LINE",line);
        end;
      rewind(1);
    end;

comment   PROCEDURE GETNODE  READS  THE INPUT FILE,  CHECKS THAT THE
         COMMAND IS A VALID 'NODE' COMMAND, AND GETS THE THREE-CHAR-
         ACTER NODE IDENTIFIER.   IT CHECKS THAT THE NODE IS UNIQUE,
         AND ASSIGNS IT A NUMBER IN THE SEQUENCE OF NODES.  THIS SE-
         QUENCE OF NAMES IS STORED IN THE ARRAY 'ORDER'.  GETNODE IS
         CALLED BY GETADDRESS;

  procedure getnode;
    begin
      string(3) node, name;
      integer bottom;
      geton(1,"A3",node);
      if node = "ODE" then
        begin
          geton(1,"A3",name);
          if ¬subflag then bottom:=1
          else bottom:=257;
          for j:=bottom until count do
            if order(j) = name then
              put(5,"A40","ERROR: NOT NEW NODE AT LINE",line);
          order(count):= name;
          count:=count+1;
        end
```

```
      else put(5,"A40","ERROR: ILLEGAL COMMAND 'N...' AT LINE",line);
    end;

comment  PROCEDURE TRANSLATE  READS THE FIRST CHARACTER OF EACH LINE
         IN THE INPUT FILE AND CALL THE APPROPRIATE HANDLING ROUTINE
         TO TRANSLATE  THE INSTRUCTION.   THIS CONTINUES  UNTIL  THE
         COMMAND 'END' IS FOUND.  CALLED BY MAIN;

  procedure translate;
    begin
      string(1) char;
      logical done;
      done := false;
      line:=1;
      get(1,"A1",char);
      if char<>"N" then
        put(5,"A40","ERROR: FIRST LINE MUST BE 'NODE ...'");
      while ¬done do
        begin
          line:=line+1;
          get(1,"A1",char);
          if char="I" then ifbranch
            else if char="G" then branch
              else if char="M" then mask
                else if char="J" then jump
                  else if char="R" then rts
                    else if char="E" then finish(done)
                      else if char="S" then sbrt
                        else if char="V" then service
                          else if char="N" then maskbit:=false
                            else put(5,"A40", "ERROR: BAD COMMAND AT LINE",line);
          if char="I" or char="G" or char="J" or char="R" then
                  control;
        end;
    end;

comment  PROCEDURE  IFBRANCH CHECKS THAT THE COMMAND IS A CORRECT IF
         COMMAND, TRANSLATES THE INPUT CONDITION AND MASK TO OUTPUT,
         GETS  THE DESTINATION NODE  AND DETERMINES  ITS ADDRESS  BY
         CALLING FINDNODE, AND TRANSLATES THE OUTPUT, IF ANY. CALLED
         BY TRANSLATE;

  procedure ifbranch;
    begin
      string(1) char;
      string(8) icon, msk, output;
      integer nadd;
      geton(1,"A1",char);
      if char = "F" then
        begin
          geton(1,"A1",char);
          while char = " " do  geton(1,"A1",char);
          for i:=1 until 8 do
            begin
              if char="X" then msk(i-1|1):=icon(i-1|1):="1"
              else if char="0" then msk(i-1|1):=icon(i-1|1):="0"
                else if char="1" then
                  begin
                    msk(i-1|1):="0";
                    icon(i-1|1):="1";
```

```
                    end
                    else put(5,"A40","ERROR: ILLEGAL INPUT AT LINE",line);
                  geton(1,"A1",char);
                end;
              put(2,"A6"," FCB %",msk);
              put(3,"A6"," FCB %",icon);
              findnode(nadd);
              put(3,"A5,I3"," FCB ",nadd);
              geton(1,"A1",char);
              if char=","  then out(output)
              else output:="00000000";
              put(2,"A6"," FCB %",output);
            end
          else put(5,"A40","ERROR: ILLEGAL IF STATEMENT AT LINE",line);
        end;

comment  PROCEDURE FINDNODE SCANS THROUGH THE ARRAY  'ORDER'  TO GET
         THE NODE'S NUMBER  IN THE SEQUENCE,  AND FINDS  THAT NODE'S
         ADDRESS BY REFERENCING THE ARRAY 'ADDRESS'.   CALLED BY SE-
         VERAL PROCEDURES;

  procedure findnode(integer result nadd);
    begin
      string(1) char;
      string(3) nod;
      integer i, limit;
      geton(1,"A1",char);
      while char=" " do  geton(1,"A1",char);
      if char="N" then
        begin
          geton(1,"A3",nod);
          if nod="ODE" then
            begin
              geton(1,"A3",nod);
              if ¬subflag then
                begin
                  i:=1;
                  limit:=256;
                end
              else
                begin
                  i:=257;
                  limit:=512;
                end;
              while order(i)¬=nod and i<limit do i:=i+1;

              if order(i)¬=nod then put(5,"A40","ERROR: ILLEGAL NODE AT LINE",line);
              nadd:=address(i);
            end
          else put(5,"A40","ERROR: BAD INSTRUCTION 'N...' AT LINE",line);
        end
      else put(5,"A40","ERROR: NO NODE AT LINE",line);
    end;

comment  PROCEDURE  BRANCH CHECKS  FOR A LEGAL  'GOTO'  INSTRUCTION,
         SETS  THE INPUT MASK AND CONDITION,  AND CALLS  FINDNODE TO
         THE ADDRESS  OF THE  DESTINATION NODE.   IT TRANSLATES  THE
         OUTPUT, IF ANY, BY CALLING OUT.  CALLED BY TRANSLATE;

  procedure branch;
```

```
      begin
        string(1) char;
        string(3) got;
        string(8) output;
        integer nadd;
        geton(1,"A3",got);
        if got="OTO" then
          begin
            findnode(nadd);
            put(2,"AB"," FCB $FF");
            put(3,"AB"," FCB $FF");
            put(3,"A5,I3"," FCB ",nadd);
            geton(1,"A1",char);
            if char="," then out(output)
            else output:="00000000";
            put(2,"A6"," FCB %",output);
          end
        else put(5,"A40","ERROR: BAD INSTRUCTION 'G...' AT LINE",line);
      end;

comment  PROCEDURE OUT  CHECKS THAT THE OUTPUT INSTRUCTION  IS LEGAL
         AND TRANSLATES THE OUTPUT TO THE STRING  'OUTPUT'  WHICH IS
         PASSED BACK  TO THE CALLING PROCEDURE.   CALLED BY  SEVERAL
         PROCEDURES;

  procedure out(string(8) result output);
    begin
      string(1) char;
      string(2) outp;
      geton(1,"A1",char);
      while char=" " do  geton(1,"A1",char);
      if char="O" then
        begin
          geton(1,"A2",outp);
          if outp="UT " then
            begin
              geton(1,"A1",char);
              while char=" " do  geton(1,"A1",char);
              for i:=1 until 8 do
                begin
                  if char="O" or char="1" then output(i-1|1):=char
                  else put(5,"A40","ERROR: ILLEGAL OUTPUT AT LINE",line);
                  geton(1,"A1",char);
                end;
            end
          else put(5,"A40","ERROR: BAD INSTR 'O..' AT LINE",line);
        end
      else put(5,"A40","ERROR: EXPECTED 'OUT' (NOT FOUND) AT LINE",line);
    end;

comment  PROCEDURE FINISH  CHECKS THAT THE COMMAND  IS A LEGAL 'END'
         COMMAND AND RETURNS THE LOGICAL 'DONE' TO PROCEDURE TRANS-
         LATE TO SIGNIFY THE END OF THE INPUT;

  procedure finish(logical result done);
    begin
      string(2) nd;
      geton(1,"A2",nd);
      if nd="ND" then
        put(5,"A40","ERROR: BAD INSTRUCTION 'E...' AT LINE",line);
```

```
      done:=true;
    end;

comment  PROCEDURE MASK  CHECKS FOR  A LEGAL MASK  INSTRUCTION,  AND
         SETS THE LOGICAL MASKFLAG TO TRUE IF SO.   CALLED BY TRANS-
         LATE;

  procedure mask;
    begin
      string(3) msk;
      geton(1,"A3",msk);
      if msk="ASK" then maskbit:=true
      else put(5,"A40","ERROR: BAD INSTRUCTION 'M...' AT LINE",line);
    end;

comment  PROCEDURE JUMP  CHECKS FOR A LEGAL  'JUMP'  COMMAND,  CALLS
         FINDNODE TO TRANSLATE  THE DESTINATION NODE ADDRESS,  CALLS
         FINDNODE AGAIN  TO TRANSLATE  THE RETURN ADDRESS,  AND SETS
         THE OUTPUT BY CALLING 'OUT' IF NECESSARY.  CALLED BY TRANS-
         LATE;
  procedure jump;
    begin
      string(1) char;
      string(3) jmp;
      string(5) return;
      integer nadd, ladd;
      string(8) output;
      geton(1,"A3",jmp);
      if jmp="UMP" then
        begin
          if ¬subflag then
            begin
              put(2,"A6"," FCB 0");
              subbit:=true;
              maskbit:=true;
              subflag:=true;
              findnode(nadd);
              subflag:=false;
              geton(1,"A1",char);
              while char=" " do  geton(1,"A1",char);
              if char="R" then
                begin
                  geton(1,"A5",return);
                  if return="ETURN" then
                    begin
                      findnode(ladd);
                      put(3,"A5,I3"," FCB ",ladd);
                      put(3,"A5,I3"," FCB ",nadd);
                      geton(1,"A1",char);
                      if char="," then out(output)
                      else output:="00000000";
                      put(2,"A6"," FCB %",output);
                    end
                  else put(5,"A40","ERROR: BAD COMMAND 'R...' AT LINE",line);
                end
              else put(5,"A40","ERROR: NO RETURN INSTR. AT LINE",line);
            end
          else put(5,"A40","ERROR: 'JUMP' IN SUBROUTINE SECN AT LINE",line)
        end
      else put(5,"A40","ERROR: BAD INSTR 'J...' AT LINE",line);
```

```
    end;

comment 'PROCEDURE RTS  CHECKS FOR A LEGAL  'RTS'  INSTRUCTION,  AND
         OUTPUTS THE INPUT CONDITION, MASK, OUTPUT, AND NEXT ADDRESS
         AND SETS  THE SUBBIT TO TRUE TO INDICATE A RETURN FROM SUB-
         ROUTINE.  CALLED BY TRANSLATE;
  procedure rts;
    begin
      string(1) char;
      string(2) ret;
      string(8) output;
      geton(1,"A2",ret);
      if ret="TS" then
        begin
          if subflag then
            begin
              put(2,"A6"," FCB 0");
              put(3,"A6"," FCB 0");
              put(3,"A6"," FCB 0");
              maskbit:=true;
              subbit:=true;
              geton(1,"A1",char);
              if char="," then out(output)
              else output:="00000000";
              put(2,"A6"," FCB %",output);
            end
          else put(5,"A40","ERROR: 'RTS' NOT IN SUBROUT SECN AT LINE",line)
        end
      else put(5,"A40","ERROR: BAD INSTR 'R...' AT LINE",line);
    end;

comment  PROCEDURE SERVICE  CHECKS FOR A LEGAL  'VECTOR ...' COMMAND
         AND CALLS FINDNODE  TO DETERMINE THE DESTINATION ADDRESS IN
         IN CASE OF INTERRUPTS.  IT PLACES  THIS ADDRESS  IN GLOBAL
         VARIABLE 'SERVADD'.   IT CALLS OUT  IF NECESSARY TO SET THE
         OUTPUT BYTE OF THE VECTOR;

  procedure service;
    begin
      string(1) char;
      string(16) handle;
      geton(1,"A16",handle);
      if handle="ECTOR INTERRUPTS" then
        begin
          if subflag then
            begin
              findnode(servadd);
              geton(1,"A1",char);
              if char="," then out(int_out);
            end
          else put(5,"A40","ERROR: INTERRUPT INSTR NOT IN SUBR SECN AT LINE",line)
        end
      else put(5,"A40","ERROR: BAD COMMAND '...' AT LINE",line);
    end;

comment  PROCEDURE SBRT CHECKS FOR A LEGAL 'SUBROUTINES' COMMAND AND
         SETS SUBFLAG TO TRUE  IF SO.   IT ALSO OUTPUTS  THE HEADERS
         FOR THE SUBROUTINE SECTIONS OF THE THREE PROM FILES. CALLED
         BY TRANSLATE;
```

```
    procedure sbrt;
      begin
        string(10) sub;
        geton(1,"A10",sub);
        if sub="UBROUTINES" then
          begin
            put(2,"A9"," ORG $200");
            put(3,"A9"," ORG $200");
            put(4,"A9"," ORG $100");
            subflag:= true;
          end
      end;

  comment  PROCEDURE CONTROL PUTS A BYTE OF INFORMATION  INTO THE PROM
           FILE 4 AT THE END OF EACH INSTRUCTION PROCESSED.   THIS IN-
           FORMATION INCLUDES THE INTERRUPT MASK,  SUBROUTINE BIT, AND
           I/O CONTROL BITS.  CALLED BY TRANSLATE;

    procedure control;
      begin
        string(8) byte;
        if maskbit or subflag then byte(0|1):="0"
        else byte(0|1):="1";
        if subbit  then byte(1|1):="1"
        else byte(1|1):="0";
        byte(2|6):="000000";
        subbit:=false;
        put(4,"A6"," FCB %",byte);
      end;

  comment  PROCEDURE  SETSERV  OUTPUTS  THE HEADERS  FOR THE INTERRUPT
           VECTOR LOCATED  AT THE  LAST BYTES  OF THE PROM FILES.   IT
           THEN INSERTS AN UNCONDITIONAL BRANCH (GOTO)  TO THE ADDRESS
           'SERVADD' CALCULATED DURING TRANSLATE.  CALLED BY MAIN;

    procedure setserv;
      begin
        put(2,"A9"," ORG $3FE");
        put(2,"A8"," FCB $FF");
        put(2,"A6"," FCB %",int_out);
        put(2,"A4"," END");
        put(3,"A9"," ORG $3FE");
        put(3,"A8"," FCB $FF");
        put(3,"A5,I3"," FCB ", servadd);
        put(3,"A4"," END");
        put(4,"A9"," ORG $1FF");
        put(4,"A6"," FCB 0");
        put(4,"A4"," END");
      end;

  comment

  ******************************
  ******************************
  ***                        ***
  ***  MAIN PROGRAM BEGINS   ***
  ***                        ***
  ******************************
  ******************************
  ;
```

```
  begin
    initialise;
    getaddress;
    translate;
    setserv;
  end;
end.
```

APPENDIX 3. Assembler I/O.

a) Low-level language input program.

```
NODE L1
MASK
IF XXX01XX1   NODE L1,  OUT  10100101
GOTO NODE L2,         OUT00111100
NODE L2
IF 10XX1X10,NODE L3, OUT 00010110
GOTO NODE L3
NODE L3
JUMP     NODE S0   RETURN   NODE L2,  OUT 11111111
SUBROUTINES
NODE S0
IF XXXXXXX0, NODE S1,         OUT 11110000
GOTO NODE S2
NODE S1
RTS
NODE S2
GOTO NODE S3
NODE S3
GOTO NODE S0
VECTOR INTERRUPTS    NODE S1,      OUT  11101110
END
```

b) MC6809 assembler language output program for PROM1 (input mask and output words).

```
ORG 0
FCB %11100110
FCB %10100101
FCB $FF
FCB %00111100
FCB %00110100
FCB %00010110
FCB $FF
FCB %00000000
FCB 0
FCB %11111111
ORG $200
FCB %11111110
FCB %11110000
FCB $FF
FCB %00000000
FCB 0
FCB %00000000
FCB $FF
FCB %00000000
FCB $FF
FCB %00000000
ORG $3FE
FCB $FF
FCB %11101110
END
```

c) MC6809 assembler language output program for PROM2 (input condition and new address).

```
ORG 0
FCB %11101111
FCB   0
FCB $FF
FCB   2
FCB %10111110
FCB   4
FCB $FF
FCB   4
FCB   2
FCB   0
ORG $200
FCB %11111110
FCB   2
FCB $FF
FCB   3
FCB 0
FCB 0
FCB $FF
FCB   4
FCB $FF
FCB   0
ORG $3FE
FCB $FF
FCB   2
END
```

d) MC6809 assembler language output program for the control word PROM.

```
ORG 0
FCB %00000000
FCB %00000000
FCB %10000000
FCB %10000000
FCB %01000000
ORG $100
FCB %00000000
FCB %00000000
FCB %01000000
FCB %00000000
FCB %00000000
ORG $1FF
FCB %00000000
END
```

APPENDIX 4. PSC circuit test programs.

a) MC6809 assembler program for load test.

```
* This program tests all possible combinations
*  of NI and the input, mask, and condition nybbles
*  and compares the result to the desired result:
*
*
* Initialise the PIA:
*
DRA     EQU  $8000      DATA AND DATA DIRECTION
DDRA    EQU  $8000        REGISTER (A SIDE).
CRA     EQU  $8001      CONTROL REGISTER.
DRB     EQU  $8002      DATA AND DIR. REG.
DDRB    EQU  $8002      (B SIDE)
CRB     EQU  $8003      CONTROL REG. (B)
*
        ORG  $DD00
*
        CLRA
        STA  CRA        ADDRESS DATA DIRECTION
        STA  CRB          REGISTERS
        LDA  #$7F       SET TOP BIT AS INPUT AND
        STA  DDRA         OTHERS AS OUTPUT.
        LDA  #$FF       SET ALL BITS AS OUTPUTS
        STA  DDRB
        LDA  #$04       ADDRESS DATA REGISTERS
        STA  CRA          ON BOTH A AND B SIDES.
        STA  CRB
*
        CLR  $DC00      CLEAR INPUT VARIABLES.
        CLR  $DC01
LOOP1   LDA  $DC00      GET NI (BIT 4) AND INPUT (3-0).
        LDB  $DC01      GET CONDITION (7-4) AND MASK (3-0)
        STA  DRA        OUTPUT TO PSC CHIP...
        STB  DRB
*
*       EVALUATE DESIRED RESULT:
*
        BITA #$10       IS NI SET HIGH?
        BNE  HIGH       YES. CHECK IF LOAD BIT IS HIGH.
```

```
*       OTHERWISE CONTINUE WITH CALCULATION
*
        ORA  $DC01      GET 'OR' OF INPUT AND MASK.
        COMA            GET 'NOR' OF INPUT AND MASK.
        LSLA            MOVE RESULT INTO
        LSLA             HIGH NYBBLE
        LSLA             OF REG. A.
        LSLA
        ORB  #$0F       SET LOW NYBBLE OF REG. B
        STB  $DC02       AND STORE IN DC02.
        EORA $DC02      GET 'EX-OR'
        INCA             IS RESULT ALL ONES?
        BNE  HIGH        NO; GO TO NEXT...
ZERO    LDA  DRA        YES; CHECK CONTROLLER
        ANDA #$80         LOAD BIT.  IS IT ZERO?
        BEQ  NEXT        YES (DESIRED RESULT).
ERR1    BRA  ERR2        NO.  ERROR IN CHIP!
*
HIGH    LDA  DRA          CHECK CONTROLLER LOAD
        ANDA #$80         BIT.  IS IT ONE?
        BNE  NEXT        YES (DESIRED RESULT).
ERR2    LDX  $DC00       NO.  ERROR IN CHIP!
        STX  $DE00     STORE FAULTY COMBINATION
*                     AND CONTINUE...
*
NEXT    LDX  $DC00        GET NEXT INPUT COMB.
        LEAX 1,X          BY INCREMENTING
        STX  $DC00        PREVIOUS ONE.
        CMPX #$FFF       ALL DONE?
        BLE  LOOP1       NO. REPEAT PROCESS.
DONE    BRA  DONE         YES; WAIT HERE.
*
        END
```

b) Low-level language program for subroutine/interrupt test.

```
NODE M0
IF XXXXXXX0  NODE M0
NODE M1
IF XXXXXX0X  NODE M2
JUMP NODE S0   RETURN NODE M3
NODE M2
JUMP NODE S3   RETURN NODE M3
NODE M3
IF XXXXXXX1  NODE M3
NODE M4
IF XXXXXX0X  NODE M5
JUMP NODE S3   RETURN NODE M0
NODE M5
JUMP NODE S0   RETURN NODE M0
SUBROUTINES
NODE S0
GOTO NODE S1,  OUT 00000001
NODE S1
GOTO NODE S2,  OUT 00000010
NODE S2
RTS,           OUT 00000100
NODE S3
GOTO NODE S4,  OUT 00001000
NODE S4
GOTO NODE S5,  OUT 00000100
NODE S5
RTS,           OUT 00000010
VECTOR INTERRUPTS NODE S3
END
```

c) Low-level language program for greatest common divisor circuit.

```
NODE S0
IF XXXXXXX0 NODE S0, OUT 00000000
NODE S1
IF XXXXXXX1 NODE S1, OUT 00000001
NODE S2
IF XXXXXXX0 NODE S2, OUT 00000000
NODE S3
IF XXXXXXX1 NODE S3, OUT 00000010
GOTO NODE S4, OUT 00000100
NODE S4
IF XXXXX01X NODE S6, OUT 00000101
NODE S5
IF XXXXX00X NODE S6, OUT 00000110
GOTO NODE S0, OUT 00000000
NODE S6
GOTO NODE S4, OUT 00000100
END
```

INPUT WORD: XXXXXabc
- a: ZERO SIGNAL (ACTIVE HIGH)
- b: POSITIVE (LOW)/NEGATIVE (HIGH) SIGNAL
- c: INPUT READY SIGNAL (ACTIVE HIGH)

OUTPUT WORD: XXXXXdef
- d: MUX SELECT LINE (LOW FOR INPUTS, HIGH FOR ADDER)
- e: REGISTER B LATCH SIGNAL
- f: REGISTER A LATCH SIGNAL